UNIVERSITY OF CAMBRIDGE

Mechanising and evolving the formal semantics of WebAssembly: the Web's new low-level language

Conrad Watt

St Catharine's College

This thesis is submitted on February 5th, 2021 for the degree of Doctor of Philosophy

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This thesis does not exceed the prescribed limit of 60 000 words.

Abstract

Mechanising and evolving the formal semantics of WebAssembly: the Web's new low-level language

Conrad Watt

WebAssembly is the first new programming language to be supported natively by all major Web browsers since JavaScript. It is designed to be a natural low-level compilation target for languages such as C, C++, and Rust, enabling programs written in these languages to be compiled and executed efficiently on the Web. WebAssembly's specification is managed by the W3C WebAssembly Working Group (made up of representatives from a number of major tech companies). Uniquely, the language is specified by way of a full pen-and-paper formal semantics.

This thesis describes a number of ways in which I have both helped to shape the specification of WebAssembly, and built upon it. By mechanising the WebAssembly formal semantics in Isabelle/HOL while it was being drafted, I discovered a number of errors in the specification, drove the adoption of official corrections, and provided the first type soundness proof for the corrected language. This thesis also details a verified type checker and interpreter, and a security type system extension for cryptography primitives, all of which have been mechanised as extensions of my initial WebAssembly mechanisation.

A major component of the thesis is my work on the specification of shared memory concurrency in Web languages: correcting and verifying properties of JavaScript's existing relaxed memory model, and defining the WebAssembly-specific extensions to the corrected model which have been adopted as the basis of WebAssembly's official threads specification. A number of deficiencies in the original JavaScript model are detailed. Some errors have been corrected, with the verified fixes officially adopted into subsequent editions of the language specification. However one discovered deficiency is fundamental to the model, an instance of the well-known "thin-air problem".

My work demonstrates the value of formalisation and mechanisation in industrial programming language design, not only in discovering and correcting specification errors, but also in building confidence both in the correctness of the language's design and in the design of proposed extensions.

This thesis is dedicated to my parents, Karen and Stephen Watt;
and to my wonderful partner, Ponpavi Sangsuradej.

Acknowledgements

I gratefully thank my supervisor, Peter Sewell, for the support he has shown me during my PhD.

In the course of carrying out this work, I have received funding from the following sources:

- EPSRC studentship (1790117)
- EPSRC research grant *REMS: Rigorous Engineering for Mainstream Systems* (EP/K008528/1)
- EPSRC research grant *Semantic Foundations for Interactive Programs* (EP/N02706X/2)
- 2019 Google PhD Fellowship in Programming Technology and Software Engineering
- VeTSS/NCSC research grant *Mechanising Concurrent WebAssembly* (G105616 RI2)
- Peterhouse Research Fellowship

Contents

Chapter 1

Introduction

WebAssembly (abbreviated Wasm) is a low-level bytecode language, designed to be a natural low-level compilation target for languages such as C, C++, and Rust, thereby enabling programs written in these languages to be compiled and executed efficiently on the Web. It is the first language since JavaScript to be implemented across all major Web browsers. JavaScript was first developed in 1995 in the early years of the World Wide Web [1], at which time Web pages were almost entirely static and consisted primarily of text. In a 1996 interview, JavaScript's creator Brendan Eich expressed his hope that the language would "become ubiquitous on the Web as the favored way of gluing HTML elements and actions on them together" [2].

In the years since JavaScript's creation, the Web has evolved significantly. Developers have continued to push the envelope on computationally intensive Web site content, such as video, image processing, and 3D games. JavaScript was not created with these use-cases in mind [1]. HTML and JavaScript initially lacked many desired multimedia features, and JavaScript, as a high-level dynamic scripting language, suffered from fundamental performance deficiencies. For years, many Web sites relied on third party plugin-based technologies such as Flash, Java, Shockwave, and Silverlight in order to add functionality. This approach had significant limitations, as Web developers needed to rely on users actually installing and updating the required plugins, which were often buggy and contained security vulnerabilities.

Web browsers have shifted towards providing more functionality natively, but various proposed alternatives to JavaScript geared towards large, computationally intensive programs have struggled with a lack of cross-browser support. As a recent example, Google's Native Client (NaCl) [3] allowed Web developers to directly embed optimised platform assembly in their Web sites; this code would be sandboxed and executed in a site visitor's browser with no plugins required. However, other browser developers were unwilling to support the technology, and so such sites would only function in Google's Chrome browser.

Mozilla, another browser developer, focussed on hyper-optimising a small subset of

JavaScript, christened asm.js, which threw away most of the language's object model and dynamic behaviour, so that asm.js could in principle be compiled efficiently ahead-of-time in-browser. Web developers generated asm.js through compilation from other languages, most notably C/C++ using the Emscripten compiler [4]. A key selling point of asm.js was that, because it was a pure syntactic subset of JavaScript, in theory any browser could run the generated code, even if the browser's developers were not specifically invested in supporting the asm.js project. This was attractive to Web developers who did not wish to limit their market reach by relying on a single-browser technology such as Native Client. In practice, however, the asm.js code generated by Emscripten was so different from hand-written JavaScript that browser JavaScript engines which did not implement asm.js-specific optimisations sometimes struggled to provide acceptable performance. Chrome's V8 JavaScript engine in particular could be very memory-hungry during the parsing and compilation of regular JavaScript, and as more ambitious generated asm.js made its way onto the Web, often hundreds of thousands of lines long, V8 needed to implement new heuristics to avoid running out of memory [5].

In general, the text-based nature of JavaScript proved to be a limitation for asm.js. The time needed to parse large asm.js files was significant, even for optimised implementations. Another issue with asm.js was that it began to place pressure on the wider JavaScript specification. Adding a feature to improve the performance of asm.js without compromising its position as a subset of JavaScript could only be done by adding the feature to the JavaScript language as a whole. A key flashpoint was the "SIMD.js" proposal [6], which would have added a low-level API for CPU vector instructions to JavaScript, primarily motivated by performance wins for asm.js code. This proposed feature caused significant complications to the specification and implementations; at the time, JavaScript had seven fundamental types of values (Undefined, Null, Boolean, Number, Symbol, String, and Object), while the SIMD.js proposal would have added ten more, each representing a particular size and kind of vector value. Even a barebones prototype implementation of the SIMD.js API was reported as representing ∼10% of V8's code size [7]. The popularity of asm.js, despite these limitations, made it increasingly clear to many parties that there was a need for a new Web language, designed from the ground up as a compact, low-level compilation target.

1.1 WebAssembly

As previous efforts such as Native Client had shown, a new Web language would need the unanimous support of major industry players in order to succeed and gain adoption. The WebAssembly project was first made public on the 17th of June 2015, in a series of announcements coordinated between representatives from all major browser vendors (Google, Mozilla, Microsoft, and Apple), and several key figures in the JavaScript specifi-

cation community [8]. This commitment by the browser vendors to collaboratively design and unanimously implement WebAssembly represented an exceptional political success. Today, WebAssembly is available in all major browsers, and its specification is managed by the WebAssembly Working Group and the wider WebAssembly Community Group, both W3C standards committees which contains representatives from all major browser vendors and a number of other tech companies. WebAssembly is distinguished from the vast majority of industry languages in that its normative specification was developed from the outset as a formal semantics.

This thesis describes my research into, and contributions to, the WebAssembly language. My work demonstrates the value of formalisation and mechanisation in industrial programming language design, not only in discovering and correcting specification errors, but also in building confidence both in the correctness of the language's design and in the design of proposed extensions.

Mechanisation In Chapter 3 I describe my mechanisation of WebAssembly's formal semantics, including a proof of soundness for the WebAssembly type system. This work was carried out during the drafting of the WebAssembly specification, and I identified and corrected several errors which originally caused the type system to be unsound. I also detail an executable verified interpreter and type checker which were built on top of the mechanisation. The chapter draws from a previously published paper *Mechanising and Verifying the WebAssembly Specification* (CPP 2018) [9], of which I was the sole author.

CT-Wasm In Chapter 4 I describe the mechanisation and verification of a proposed extension to WebAssembly's type system which is designed to improve the resistance of cryptographic algorithms to timing attacks. The chapter draws from a previously published paper *CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem* (POPL 2019) [10], authored by myself, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. My co-authors developed the initial unformalised design for the type system which was the basis for my formalisation and mechanised proof, and created a number of related concrete implementations and tools.

Relaxed memory In Chapter 5 I describe my work on the relaxed memory models (concurrent semantics) of JavaScript and WebAssembly. WebAssembly's memory model is heavily based on an existing feature of JavaScript: the SharedArrayBuffer. I discovered and corrected a number of errors in JavaScript's memory model, including a failure of the model to support an intended compilation scheme to Armv8-A, and a violation of the model's stated correctness condition — a variant of Sequential Consistency for Data Race Free programs (SC-DRF). My proposed fixes to the model are supported by finite model checking in Alloy [11] and mechanised proof in Coq [12]. Other outstanding issues

with the model are identified and placed in the context of known problems in the field of relaxed memory. Then, the WebAssembly-specific extensions to the model are described. The described relaxed memory model has been officially adopted as part of the in-progress WebAssembly threads specification.

This work was carried out with a number of collaborators. Christopher Pulte developed a model and tooling for Armv8-A executions which were used in verifying that the relevant changes to the JavaScript model were correct. This model is included as Appendix A. Anton Podkopaev developed a compilation scheme correctness proof in Coq for a subset of the model to a number of other architectures, to improve confidence in our model changes. Guillaume Barbier assisted with the execution of Alloy searches and some exploratory mechanisation of the JavaScript model, as part of a student internship with me. Stephen Dolan participated in initial discussions around proposed fixes for the JavaScript model, and suggested the first concrete example which demonstrated the Armv8-A compilation failure. Shaked Flur assisted with tooling for the Armv8-A model. Shu-yu Guo is a member of ECMA TC39 (JavaScript's standards body) who helped to present and ratify our proposals for changes to the JavaScript model. Jean Pichon-Pharabod participated in discussions on both the JavaScript and WebAssembly models throughout the work, and implemented some exploratory SMT-based tooling for the WebAssembly model. Andreas Rossberg, the WebAssembly specification editor, assisted in formalising the WebAssembly model in a way which fit the conventions of the specification.

Several others contributed to the work. Lars T Hansen, another member of ECMA TC39, shared with us important historical context on the development of the relaxed memory model. Hans Boehm and Ori Lahav both independently made us aware of an example which demonstrated a weakness in JavaScript's statement of SC-DRF beyond the violation mentioned above.

The chapter draws from two previously published papers: *Weakening WebAssembly* (OOPSLA 2019) [13, 14], authored by myself, Jean Pichon-Pharabod, and Andreas Rossberg, and *Repairing and Mechanising the JavaScript Relaxed Memory Model* (PLDI 2020) [15], authored by myself, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo.

Supplemental materials The Isabelle/HOL, Coq, and Alloy code associated with this thesis can be found at `https://github.com/conrad-watt/wasm-thesis-aux` [16].

Chapter 2

Background

2.1 WebAssembly design

WebAssembly is a bytecode language supported by all major Web browsers (at the time of writing, Chromium, Firefox, and Safari). Its initial feature set aims to be a natural and efficient compilation target for low-level languages with manual memory management such as C/C++ and Rust. This chapter will introduce the key concepts and definitions of the WebAssembly language and its official formal semantics. WebAssembly's formal semantics was originally presented as an academic paper by Haas et al. [17], who list four core design principles of WebAssembly:

Safety The modern Web is inherently a platform for running untrusted code on a user's computer. In pursuing efficiency, WebAssembly cannot compromise the Web's existing security model; the language is designed to give no more capabilities to an attacker than were already available through JavaScript. While this does mean that WebAssembly must incur some overhead in additional dynamic checks, often these checks can be avoided either through static type system restrictions, or clever implementation strategies (§5.8.1 will later discuss one such strategy in a concurrent context).

Speed WebAssembly has been developed in response to the growth of heavy-duty, low-level client-side computation on the Web. Its operations are kept as close as possible to the real behaviour of platform assembly languages. Because it is intended for use within Web sites, the time taken to decode and compile the bytecode is also of significant concern — metrics such as overall "time to interactive" (the total time required for a user's Web browser to render a Web site and compile its associated scripts) are in many cases considered more important than the runtime performance of the compiled code. Because it is a bytecode language, it can be decoded more quickly than a text-based language such as asm.js can be parsed. WebAssembly is carefully structured so that decoding and compilation can be carried out in a single, combined linear pass, which can begin in a

streaming fashion while the tail of the WebAssembly code is still downloading, at a speed such that the user's internet connection can become the main bottleneck [18].

Portability Because a Web site containing WebAssembly code may be accessed by many different combinations of Web browser, operating system, and architecture, platform-specific behaviour must be avoided in WebAssembly's design wherever possible. It is generally assumed that, because of the huge number of developers and toolchains producing WebAssembly programs, even the most minor inter-platform discrepancy will be exposed by some piece of real-world code. For this reason, WebAssembly takes great pains to guarantee deterministic behaviour, with only a few small caveats related to floating-point bit patterns and resource exhaustion.

Compactness One of the key deficiencies of asm.js was the text-based nature of JavaScript, which placed fundamental limits on download and parsing speed in a setting where bandwidth use and start-up time are crucial [19]. WebAssembly's bytecode format aims to significantly reduce code file sizes compared to a text-based representation, resulting in benefits for Web hosts, users, and developers. Like the Java bytecode, it is *stack-based*. This means that common operations such as addition do not need to explicitly name their operands, resulting in a more compact representation.

Throughout Haas et al. [17], another design principle is pursued, beyond the four explicitly listed.

Formalisation WebAssembly was designed from the start using a formal semantics (albeit one that was purely pen-and-paper). Moreover, it is an official policy of the WebAssembly Working Group that new features must be fully formalised as extensions of this semantics before they can be adopted [20]. This is an great step forward for industry languages, which habitually rely on prose specifications. The unambiguous nature of WebAssembly's formal semantics helps to avoid inadvertent divergence between implementations, and facilitates the identification of edge-cases in new features during the design phase. This is especially important on the Web: the "WebCompat" principle is a broad convention that evolving Web standards must never introduce a change which "breaks the Web" (i.e. causes a previously functioning Web site/application to behave differently) [21]. Given the immeasurable amount of Web content online, it is generally assumed, and borne out by previous experience, that even the most convoluted edge-case may be relied upon somewhere, meaning that backwards-incompatible changes to fix language design mistakes are often impossible.

2.2 WebAssembly features

The following sections will explain the basic principles of WebAssembly before going into more detail about the specifics of the formal semantics.

WebAssembly is a low-level imperative language with first-order fully-applied functions, and a simple mechanism for dynamic dispatch based on dynamically checked function tables. Its operations are *stack-based*, meaning they produce and consume values from a function-local value stack rather than being explicitly applied to their operands. The language allows primitive values to be loaded from and stored in declared global variables, and serialised to and deserialised from a linear buffer of bytes called a *memory*.

WebAssembly's current design is often referred to as an MVP (Minimum Viable Product). The priority was getting a robustly designed feature-set that all browser vendors could agree to implement. The language described in this chapter is purely single-threaded, although we will discuss formalism issues related to a concurrent extension in Chapter 5. It is expected that more features will be added in the coming years, such as first-class function references.

WebAssembly code is distributed as *modules*. A module is a collection of functions, together with declarations of global state. Modules may be composed together, sharing state through a system of *imports* and *exports*. The precise formal structure of the module will be later explained in §2.5. The WebAssembly specification defines a bytecode representation for modules and their associated functions, but this thesis will only deal with WebAssembly's abstract syntax.

WebAssembly code must be *validated* before execution. Validation is a linear type-checking pass (broadly similar to the Java bytecode verifier [22]) which checks the bounds of any static indices and ensures that, at every program point, the shape of the stack will allow the current instruction to successfully execute (see §2.6). Validation can be done while decoding, as part of the language's streaming compilation model.

At a high level, a WebAssembly program will be delivered (for example, as part of a Web site) as a collection of modules, together with a host script which defines the order in which the modules should be validated and initialised, and how their imports and exports should be wired up. On the Web, the host script must be written in JavaScript, but WebAssembly is also finding use in non-Web and server-side contexts where the host script may be written in a different language.

2.2.1 Security model

WebAssembly modules explicitly declare their imports and exports, and a module cannot interfere with the execution of any other part of the system unless the host specifically passes in an import which gives that capability. This design is chiefly concerned with protecting the wider system, and other WebAssembly modules, from rogue behaviour.

However, *within* a group of WebAssembly modules which are coupled through imports, there are few security protections. For example, WebAssembly modules may declare or import a *memory* (see §2.2.4.3), which is a simple linear buffer of bytes. Accesses to this memory are through integer indices which are bounds-checked against the length of the buffer, and the memory can only be accessed by a module which explicitly declares or imports it. This ensures that accesses cannot overflow into other areas of system memory. However, if multiple modules import the same memory, there is no mechanism for one module to protect data it writes into memory from the others (i.e. no structured objects or checked pointers exist in the memory). This all-or-nothing model gives rise to some classic binary security vulnerabilities (albeit limited to within the group of coupled modules), as discussed by Lehmann et al. [23].

2.2.2 Value types

WebAssembly values are incredibly simple. There are only four fundamental types of value: 32- and 64-bit integers, and 32- and 64-bit floats.

$$\text{(value types)}\quad t ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$$

As mentioned, WebAssembly is a stack-based language, like the Java bytecode. However it differs from Java in several important ways. WebAssembly's stack contains only value types, as defined above. The MVP has no object references or first-class function references. Objects in a higher-level language compiled to WebAssembly must be explicitly laid out in WebAssembly's memory as a series of bytes, with pointers to those objects becoming bare i32 indices into the linear memory (see §2.2.4.3). Source-language function pointers are not representable directly in memory, but must be represented through an indirection as indices into a table of function closures (see §2.2.4.2).

2.2.3 Control flow

WebAssembly's intra-function control flow is *structured*. Instead of **goto**, WebAssembly has explicit **block**, **loop**, and **if** bytecodes. Any instruction in the bytecode stream that occurs between one of these opcodes and a matching **end** opcode is considered to be part of the *body* of that control construct (there is also an **else** opcode for **if**). These constructs can be targetted by the **br** instruction, which functions like a labelled break/continue statement from a higher-level language. The **block**, **loop**, and **if** opcodes are type-annotated to ensure that the structure of the stack remains predictable during validation, no matter what control flow occurs during execution. Their precise formal semantics will be explained in §2.4.4.

The choice to only support structured control flow was made to maintain the simplicity of WebAssembly's semantics, and because of concerns about the ability of some Web

engines to optimise irreducible control flow [24]. Languages with more general unstructured control flow such as C/C++ must transform their code into a structured form during compilation to WebAssembly. Since compiling these languages to asm.js already required this transformation (as JavaScript also lacks arbitrary **goto**), much of the necessary engineering work had already been done. It is likely that more complex control flow, such as exceptions, will be introduced to WebAssembly through future features.

2.2.4 Global state

Aside from pushing and popping values to/from the stack, some WebAssembly instructions may interact with "global" state, declared at the top-level of the enclosing module. A WebAssembly module can declare four main kinds of module-scoped object: *functions*, *tables*, *memories*, and *global variables*, described in turn below. A module can declare some subset of these to be exported into the environment, and may declare imports which must be satisfied by the exports of other modules.

As a general rule, WebAssembly objects are not accessed using explicit names, but through indices. For example, the instruction (**global.get** i) will get the value of the i-th declared global variable of the module. As a convention, objects imported from other modules occur first in the index space (such imports are statically declared), followed by the module's own declarations.

2.2.4.1 Functions

A WebAssembly module declares a list of functions, and a list of function imports that can be satisfied by the exports of other modules. Each function consists of a type annotation, a number of function-scoped statically typed *local variable declarations*, and the function *body* (containing WebAssembly instructions). As an implementation detail, WebAssembly modules statically declare a list of allowed function types, and then each function pre-declares an index into this list which denotes its type signature. Therefore, functions' bodies may contain arbitrary direct calls to each other, with no declaration order restrictions, while still allowing validation to be conducted as a single linear pass.

Local variables are statically declared to hold values of one of the four fundamental value types. They are statically accessed in the function body using indices. All local variables are function-scoped and their lifetime is restricted to the duration of the function call.

Functions may additionally be imported from the host environment. In a Web site script, for example, a JavaScript function may be passed into a WebAssembly module as an import. Such a function is automatically wrapped in the necessary type checking/casting code to convert JavaScript values into one of the four WebAssembly value types.

One function may be uniquely distinguished as the *start* function, which will execute immediately after the module is decoded, compiled, and initialised.

2.2.4.2 Tables

In order to support indirect calls without needing to handle arbitrary first-class functions, WebAssembly modules may declare a mutable *table* of function references/closures. WebAssembly code may indirectly call a function stored in the table at a dynamically-determined index. Moreover, the host environment is permitted to mutate the table during runtime, by inserting references to exported WebAssembly functions, or its own host functions. It is through this mechanism that function pointers/references may be implemented using an indirection in WebAssembly: the reference becomes an index into the table. The table may contain multiple functions with different types, so making an indirect call through the table is the one of the few areas of WebAssembly where a runtime type check is required.

Note that function closures cannot currently be pushed onto WebAssembly's stack, or stored in ordinary variables. The WebAssembly module can declare a list of indices into its function list which pre-populate the module's table. These are known as the *element segments*. To mutate the table during runtime, WebAssembly must rely on the host (for example, by importing and calling a JavaScript function). First-class function references for WebAssembly, and references carrying a precise function type which do not need to be dynamically checked, are planned as future features [25].

2.2.4.3 Memories

A WebAssembly module may declare a memory: an integer-indexed linear buffer of raw bytes. Typed WebAssembly stack values can be serialised to and deserialised from this memory. In order to ensure portability, there is almost no non-determinism in this process. Memory is zero-initialised, and WebAssembly has no trap representations; every combination of bytes of the appropriate length can be deserialised into a typed value. Moreover, almost every WebAssembly value has a single defined bit representation, with the unfortunate exception of NaN floating-point values, which are allowed a limited space of representations due to divergence between platforms.

The module's memory may be accessed by load and store instructions which take integer indexes, representing an offset in the memory — there is no integer-pointer distinction. All accesses are bounds checked to the length of the memory, and an out-of-bounds access immediately terminates execution. There are also operations to dynamically grow the memory at runtime. While some WebAssembly implementations must explicitly compare the memory access index with the memory's current length, many modern implementations use a strategy based on OS-level *trap handlers* to support a bounds-checking semantics

with minimal runtime cost (see §5.8.1).

Currently, a WebAssembly module can only declare a single table and a single memory. However, an imminent extension to the language allows modules to declare multiple tables and multiple memories.

2.2.4.4 Globals

Finally, a module may declare a number of global variables with statically-associated WebAssembly value types. These behave similarly to function-local variables, except they are accessible to all code in the module, and have their own distinguished index space. Moreover, global variables may be declared *immutable*.

2.3 WebAssembly semantics – conventions

As previously mentioned, the semantics of WebAssembly is fully formalised, initially by Haas et al. [17], and later as part of the official W3C specification [26]. Throughout this thesis, this formal specification will be referred to as the *paper semantics/specification* (in the sense of "pen-and-paper"). The relevant core of this specification is introduced and discussed over the next few sections. As we will see, the paper specification makes use of several syntactic tricks to appear more concise or aesthetically pleasing, which must be disambiguated in the mechanisation (see §3.2). The WebAssembly Community Group also maintains a "living specification", which contains candidate specification text and formal semantics for upcoming features [27]. The formalism presented below does not contain these additions, as they may be subject to further iteration.

The execution of WebAssembly code is specified using a small-step reduction relation. The top-level form of this relation is as follows:

$$S; F; e^* \hookrightarrow S; F; e^*$$

where:

$$\begin{array}{lrl} \text{(store)} & S & ::= \{\ \text{funcs} :: \ldots,\quad \text{tabs} :: \ldots,\quad \text{mems} :: \ldots,\quad \text{globs} :: \ldots\ \} \\ \text{(frame)} & F & ::= \{\ \text{locs} :: \ldots,\quad \text{inst} ::= \ldots\ \} \\ \text{(instruction)} & e & ::= \quad \ldots \end{array}$$

S is the *store*; a record containing all global state declared and used across any executing module. This state consists of functions, tables, memories, and global variables. F is the function-local *frame*, which contains the current values of declared local variables (including the function arguments), and the *instance*, which keeps track of which elements of the global store are in scope for the currently executing code. The e^* component is the list of instructions/expressions currently being executed. The tuple $S; F; e^*$ is the runtime

state, and is referred to as a *configuration*. The full internal details of these components, including the types of the store and frame record fields, will be described later in this chapter.

2.3.1 The value stack

WebAssembly is a stack-based language. Stack values have one of four *value types*: the (32- and 64-bit) integer types i32 and i64, and the (32- and 64-bit) floating-point types f32 and f64. WebAssembly's stack is much simpler than conventional ISAs such as x86 assembly. It does not hold return addresses or function pointers, only pure values of one of the four mentioned types. A "stack" in the context of WebAssembly always refers to the value stack. There is no explicit control stack or function call stack.

$$\text{(value types)} \quad t ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$$

Sometimes, $|t|$ is written to represent the size of a given type in *bytes*. For example, $|\mathsf{i32}| = 4$.

To show how WebAssembly's formalism represents the stack, we introduce two simple WebAssembly instructions: (**i32.const** k), which pushes a constant i32 value k onto the stack, and (**i32.add**), which pops two i32 values, and pushes the result of their addition. Here is a reduction rule for (**i32.add**) in the style of the specification (by convention, the store and frame are elided, as they are unaltered).

$$(\textbf{i32.const}\ j)(\textbf{i32.const}\ k)(\textbf{i32.add}) \hookrightarrow (\textbf{i32.const}\ (j +_{\mathsf{i32}} k))$$

The (**i32.const** k) instruction has no defined reduction rule on its own. Instead, reduction rules are directly defined on a leading list of (**i32.const** k) instructions together with the operation to be executed. This list of **const** instructions represents the current state of the stack. Reduction rules are defined over the just the relevant **const** list being manipulated by the currently executing instruction. Later, we will introduce congruence rules and evaluation contexts defined by the formalism which enable these reduction rules to be generalised to larger stacks and program contexts.

2.3.1.1 Notation convention: lists and options

Given a metatheoretic type *typ*, *typ** represents the type of a list of *typ* values, while *typ*$^+$ represents the type of a non-empty list.

Given a list *lst*, *lst*$[x]$ represents the element at index x of the list. Moreover, *lst*[x..y) denotes the sublist of *lst* in the range x (inclusive) to y (exclusive). Implicitly, when such list operations are part of a premise or side-condition of an inference rule, the rule cannot be applied if any index is out of bounds. The length of the list is represented by $|lst|$. The empty list is written as ϵ.

Because of their prevalence, list construction, concatenation, and append are often represented without explicit symbols, merely using juxtaposition. For example, the previously shown code fragment (**i32.const** j)(**i32.const** k)(**i32.add**) is treated as a list of instructions with three elements.

An option type is represented as $typ^?$. In the paper specification, when an option is not present, it is usually omitted entirely instead of explicitly writing none or similar. To be less ambiguous, this thesis will use the symbol "$-$" to represent none.

2.3.2 Function types

WebAssembly code fragments can be assigned a type of the form

$$\text{(function type)} \quad ft ::= t^* \to t^*$$

This type decribes how the (implicit) value stack is transformed by the execution of the instruction. The left component of the type is a list of value types representing the shape of the stack before execution, while the right component represents the resulting shape of the stack after execution.

For example, the (**i32.const** j) instruction can be given the type $\epsilon \to [\mathsf{i32}]$, and also the type $[\mathsf{i32}] \to [\mathsf{i32}, \mathsf{i32}]$, in both cases representing the pushing of one value to the stack. The (**i32.add**) instruction can be given the type $[\mathsf{i32}, \mathsf{i32}] \to [\mathsf{i32}]$, and the code fragment (**i32.const** j) (**i32.const** k) (**i32.add**) can be given the type $\epsilon \to [\mathsf{i32}]$. Some control constructs that we will see later are explicitly type-annotated to ensure that the shape of the stack remains predictable, regardless of control flow. We will see below how these types are used to define WebAssembly's type system. First, we will introduce the AST definitions and runtime semantics of WebAssembly's instructions.

2.3.3 Further conventions of the WebAssembly formalism

2.3.3.1 Types and terms

The paper semantics makes a habit of punning metavariables for types with their instances. For example, consider the following (simplified) congruence rule:

$$\frac{S; F; e^* \hookrightarrow S'; F'; e'^*}{S; F;\ e^*\ e''^* \hookrightarrow S'; F';\ e'^*\ e''^*}$$

The premise of the rule shows a configuration reducing one step. The symbol e^* represents some arbitrary list of expressions, while e'^* represents some other list of expressions which may be entirely distinct, including having a different length. The first configuration $S; F; e^* e''^*$ in the conclusion of the rule has the same components as the initial configuration in the premise, except that another arbitrary list of expressions e''^*

has been concatenated on the right of the initial expression list. In sum, this rule says that if a configuration can run one step, an otherwise identical configuration with additional trailing instructions can run the same step, and the trailing instructions will remain unchanged.

Sometimes a rule must explicitly refer to the length of a list. The symbol lst^n represents a list term of length n.

2.3.3.2 Values

As mentioned, the type-annotated **const** instruction, a term of metatheoretic type e, is also used to represent values on the stack during reduction. The metavariable v is used to refer to some **const** instruction. For example, consider the following congruence rule:

$$\frac{S; F; e^* \hookrightarrow S'; F'; e'^*}{S; F;\ v^*\ e^* \hookrightarrow S'; F';\ v^*\ e'^*}$$

This says that if a configuration can run one step, the same configuration with a "deeper" stack can run the same step, with that part of the stack remaining unchanged.

2.3.3.3 Records

Some parts of WebAssembly's state are represented using records. For example, global variables are defined as follows:

$$\text{(mutability)} \quad mut ::= \quad \mathsf{mut} \mid \mathsf{immut}$$

$$\text{(global)} \qquad glob ::= \{\ \text{mut} :: mut, \quad \text{val} :: v\ \}$$

This also shows an example of punning between types and terms occurring in the other direction: the val field is restricted to e terms of the form $t.\mathsf{const}$.

The notation for a record term is similar:

$$my\_glob = \{\text{mut}\ \ \mathsf{mut}, \quad \text{val}\ \ (\mathbf{f64.const}\ 0)\ \}$$

This represents a mutable f64 global variable with current value 0.

Given a record rec, $rec.\text{fld}$ represents the value of the relevant field. The term $rec[\text{fld} := val]$ represents the functional update of the record rec, with the relevant field updated to the stated value. Finally, $\{\text{fld}\ val^*\} \oplus rec$, where fld is a field of rec with a list type, represents a record with the same fields as rec, except fld has val^* concatenated on the left.

2.4 Instruction semantics

In this section we will explain each WebAssembly instruction in turn, describing each instruction's associated reduction rules and relevant formal definitions.

2.4.1 Stack operations

2.4.1.1 Basic operations

$$
\begin{array}{ll}
\text{(instructions)} & e ::= t.\mathbf{const}\ k \mid \mathbf{nop} \mid \mathbf{drop} \mid \mathbf{select} \mid \mathbf{unreachable} \mid \ldots \\
\text{(administrative instructions)} & e ::= \ldots \mid \mathbf{trap} \mid \ldots
\end{array}
$$

$$
\text{(values)} \quad v ::= t.\mathbf{const}\ k
$$

$$
\begin{array}{ll}
\text{(constants)} & j, k, l ::= \ldots \\
\text{(value types)} & t ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}
\end{array}
$$

$$
\begin{array}{rcll}
(\mathbf{nop}) & \hookrightarrow & \epsilon & \\
v\ (\mathbf{drop}) & \hookrightarrow & \epsilon & \\
v_1\ v_2\ (\mathbf{i32.const}\ b)(\mathbf{select}) & \hookrightarrow & v_1 & \text{where } b \neq 0 \\
v_1\ v_2\ (\mathbf{i32.const}\ b)(\mathbf{select}) & \hookrightarrow & v_2 & \text{where } b = 0 \\
(\mathbf{unreachable}) & \hookrightarrow & (\mathbf{trap}) &
\end{array}
$$

The **const** instruction has no reduction rule. As noted already, the reduction rules directly produce and consume **const** instructions, instead of explicitly modelling a separate value stack. The WebAssembly specification simply defines values using **const** instructions, and makes use of the two interchangeably, effectively punning between the type of "values", and the instructions restricted to **const**. By convention, the metavariable v denotes not only the type of values, but also indicates a **const** instruction in a reduction rule.

The **nop** instruction does nothing.

The **drop** instruction discards the top value on the stack. Note that this is the first example of the metavariable v being used to represent a **const** instruction of some (unimportant) type.

The **select** instruction functions as a ternary operator, taking three arguments from the stack. The first two arguments are values which will be selected between, while the third value is the condition. WebAssembly uses i32 types as its booleans, with a non-zero value indicating true and a zero value indicating false. If the condition value is non-zero, the first argument is returned. If the condition value is zero, the second argument is returned. The type system (see §2.6) enforces statically that both values will be of the same type.

The **unreachable** instruction is WebAssembly's version of `assert(false)`. Executing it immediately terminates execution with an error. This is represented by a special **trap** result, our first example of an *administrative instruction*. In the WebAssembly formalisation, administrative instructions represent intermediate expressions which are needed to specify reduction, but do not occur in the concrete syntax of the language. The **trap** administrative instruction represents an unrecoverable error which will immediately stop execution. We will see other examples later.

2.4.1.2 Unary operations

$$\text{(instructions)}\quad e ::= \ldots \mid t.unop_t \mid t.testop_t \mid \ldots$$

$$\begin{aligned} unop_{\mathsf{i}N} &::= \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt} \\ unop_{\mathsf{f}N} &::= \mathsf{neg} \mid \mathsf{abs} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \\ &\quad\ \ \mathsf{trunc} \mid \mathsf{nearest} \mid \mathsf{sqrt} \\ testop_{\mathsf{i}N} &::= \mathsf{eqz} \end{aligned}$$

$$\begin{aligned} (t.\mathsf{const}\ j)(t.unop_t) &\hookrightarrow (t.\mathsf{const}\ k) && \text{where } unop_t(j) = k \\ (t.\mathsf{const}\ j)(t.testop_t) &\hookrightarrow (\mathsf{i32.const}\ 1) && \text{where } testop_t(j) = \mathsf{true} \\ (t.\mathsf{const}\ j)(t.testop_t) &\hookrightarrow (\mathsf{i32.const}\ 0) && \text{where } testop_t(j) = \mathsf{false} \end{aligned}$$

The (type-annotated) unary instructions $t.unop_t$ consume one value, apply their operation, and return the result, which is always of the same type. Note that the allowed operations are different for float and integer types, and the slight (standard) abuse of notation where a syntactic component of the operation is identified with the semantic function carrying out the operation (e.g. the $unop_t$ of $(t.unop_t)$). There is some mild dependent typing in the definitions, since for an instruction $(t.unop_t)$ the value of t will determine whether $unop_t$ refers to the integer unops $unop_{\mathsf{i}N}$ or the float unops $unop_{\mathsf{f}N}$. The effects of the operations are exhaustively formalised in the official WebAssembly specification, but none of the results presented in this thesis will rely on their precise definitions, which will be omitted here. In brief, the integer operations are the bit operations "count leading zeroes", "count trailing zeroes", and "set bit/population count", while the floating point operations are standard negation, absolute value, rounding with different modes, and square root.

The test operations consume one value, and perform a boolean test, returning an i32 representing the result of the test (1 for true, 0 for false). For now, only a single test operation is defined, that being "is equal to zero" for integer types.

2.4.1.3 Arithmetic and binary operations

$$\begin{aligned} &\text{(instructions)} \quad e ::= \ldots \mid t.binop_t \mid t.relop_t \mid \ldots \\ &\text{(signedness)} \quad sx ::= \mathsf{s} \mid \mathsf{u} \end{aligned}$$

$$\begin{aligned} binop_{\mathsf{i}N} &::= \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div\_}sx \mid \mathsf{rem\_}sx \mid \mathsf{or} \mid \\ &\quad\ \ \mathsf{and} \mid \mathsf{xor} \mid \mathsf{shl} \mid \mathsf{shr\_}sx \mid \mathsf{rotl} \mid \mathsf{rotr} \\ binop_{\mathsf{f}N} &::= \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \\ &\quad\ \ \mathsf{min} \mid \mathsf{max} \mid \mathsf{copysign} \end{aligned}$$

$$\begin{aligned} relop_{\mathsf{i}N} &::= \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt\_}sx \mid \mathsf{gt\_}sx \mid \\ &\quad\ \ \mathsf{le\_}sx \mid \mathsf{ge\_}sx \\ relop_{\mathsf{f}N} &::= \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge} \end{aligned}$$

$$
\begin{array}{lcll}
(t.\mathbf{const}\ j)(t.\mathbf{const}\ k)(t.binop_t) & \hookrightarrow & (t.\mathbf{const}\ l) & \text{where } binop_t(j,k) \ni l \\
(t.\mathbf{const}\ j)(t.\mathbf{const}\ k)(t.binop_t) & \hookrightarrow & (\mathbf{trap}) & \text{where } binop_t(j,k) = \bot \\
(t.\mathbf{const}\ j)(t.\mathbf{const}\ k)(t.relop_t) & \hookrightarrow & (\mathbf{i32.const}\ 1) & \text{where } relop_t(j,k) = \mathsf{true} \\
(t.\mathbf{const}\ j)(t.\mathbf{const}\ k)(t.relop_t) & \hookrightarrow & (\mathbf{i32.const}\ 0) & \text{where } relop_t(j,k) = \mathsf{false}
\end{array}
$$

The binary instructions consume two values of type t, apply their operation, and produce a value of the same type. Unlike unary instructions, binary instructions may fail irrecoverably (for example, dividing by zero). This is represented by the case where the instruction reduces to **trap**. Moreover, some floating point operations, when returning NaN results, are specified as non-deterministically returning one of a number of permitted NaN bit representations, over-approximating the divergent behaviour of real hardware. This is modelled by the side-condition $binop_t(j,k) \ni l$, treating the possible results as a set which is arbitrarily picked from.

The relation instructions consume two values of the same type and perform a comparison, returning an i32-encoded boolean representing the result.

Some operations are annotated with a *signedness*, to denote that the operation has a signed (s) and unsigned (u) variant. WebAssembly integers are treated mostly in an unsigned fashion, but operations marked as signed will instead transiently interpret their bit representation as two's complement. This choice was made to avoid requiring two integer types for each bit-width. Signed 32-bit integers in a source language such as C are compiled to Wasm's i32 type, and arithmetic operators annotated with s where necessary to preserve the source semantics.

2.4.1.4 Type conversion

$$
\begin{array}{rcl}
\text{(instructions)} & e & ::= \ldots \mid t.cvtop_{(t,sx^?)} \mid \ldots \\
& cvtop & ::= \mathbf{convert} \mid \mathbf{reinterpret}
\end{array}
$$

$$
\begin{array}{lcll}
(t_1.\mathbf{const}\ j)(t_2.cvtop_{(t_1,sx^?)}) & \hookrightarrow & (t_2.\mathbf{const}\ k) & \text{where } cvtop_{(t_1,t_2,sx^?)}(j) = k \\
(t_1.\mathbf{const}\ j)(t_2.cvtop_{(t_1,sx^?)}) & \hookrightarrow & \mathbf{trap} & \text{where } cvtop_{(t_1,t_2,sx^?)}(j) = \bot
\end{array}
$$

This group of instructions deals with casting. The **convert** operation denotes casts which change representation. For example, $\mathbf{i32.convert}_{(\mathsf{f64},-)}$ converts a 64-bit float value to a 32-bit integer by (value) truncation. The operation will fail with a **trap** if the truncated integer value is not representable in 32 bits. Similarly, $\mathbf{i32.convert}_{(\mathsf{i64},\mathsf{s})}$ converts a 32-bit integer to a 64-bit one. The sign parameter s means that the value will be sign-extended (two's complement representation). With parameter u, the most significant bits are set to 0. Nonsensical combinations of type and sign parameter (e.g. a sign extension parameter when converting to float) are forbidden at decode-time.

The **reinterpret** operation denotes casts which change the type of the value while leaving the binary representation identical. The operation is only (syntactically) permitted to cast between values of the same size (i.e. between i32 and f32, or i64 and f64). This is enforced at decode-time.

2.4.2 Local and global variables

$$\text{(immediates)} \quad i ::= \mathit{int32}$$

$$\text{(instructions)} \quad e ::= \ldots \mid \textsf{local.get}\ i \mid \textsf{local.set}\ i \mid \textsf{local.tee}\ i \mid \textsf{global.get}\ i \mid \textsf{global.set}\ i \mid \ldots$$

$$\text{(mutability)} \quad \mathit{mut} ::= \textsf{mut} \mid \textsf{immut}$$

$$\begin{array}{lll}
\text{(global)} & \mathit{glob} ::= \{\ \text{mut} :: \mathit{mut},\quad \text{val} :: v\ \} & \\
\text{(store)} & S ::= \{\ \text{funcs} :: \ldots,\quad \text{tabs} :: \ldots,\quad \text{mems} :: \ldots,\quad \text{globs} :: \mathit{glob}^*\ \} & \\
\text{(instance)} & \mathit{inst} ::= \{\ \text{types} :: \mathit{ft}^*,\quad \text{ifuncs} :: i^*,\quad \text{itabs} :: i^*,\quad \text{imems} :: i^*,\quad \text{iglobs} :: i^*\ \} & \\
\text{(frame)} & F ::= \{\ \text{locs} :: v^*,\quad \text{inst} ::= \mathit{inst}\ \} &
\end{array}$$

$$\begin{array}{rcll}
S; F; (\textsf{local.get}\ k) & \hookrightarrow & S; F; (F.\text{locs})[k] & \\
S; F; v\ (\textsf{local.set}\ k) & \hookrightarrow & S; F'; \epsilon & \text{where } F' = F \text{ with } (\text{locs}[k]) = v \\
S; F; v\ (\textsf{local.tee}\ k) & \hookrightarrow & S; F; v\ v\ (\textsf{local.set}\ k) & \\
S; F; (\textsf{global.get}\ k) & \hookrightarrow & S; F; (S.\text{globs})[j].\text{val} & \text{where } j = (F.\text{inst.iglobs})[k] \\
S; F; v\ (\textsf{global.set}\ k) & \hookrightarrow & S'; F; \epsilon & \text{where } j = (F.\text{inst.iglobs})[k] \\
 & & & S' = S \text{ with } S'.\text{globs}[j] = v
\end{array}$$

These instructions deal with manipulating WebAssembly's local and global variables. The reduction rules for these instructions deal with new parts of WebAssembly's formalism.

To recap, the store S holds the global state of all WebAssembly code in the environment. The frame F holds local state related to the current function call. The instance *inst*, a component of the frame, keeps track of which elements of the global store are in scope for the currently executing code. As previously mentioned, a module is currently only permitted to reference at most one memory and one table, but the formal definition of the instance is future-proofed to support multiples of each, represented as lists.

Local variables are declared on a per-function granularity (see §2.4.5). Global variables are declared on a per-program/module granularity (see §2.5). Therefore, the current values of local variables are held in the frame, while the current values of global variables are held in the store, with an indirection through the current instance (see below).

Local variable accesses are performed using a static index. The instruction (**local.get** 3) will return the value of the third local variable in the current frame. Similarly, the instruction (**local.set** 2) will remove one value from the stack, and set it as the value of the second local variable in the frame. The **local.tee** instruction acts as **local.set**, except the value from the stack is not removed, but left on the stack (effectively copied). It is equivalent to the sequence (**local.set** k)(**local.get** k), and is provided to reduce the code size of this common pattern.

Global variable accesses are also performed using a static index. Because multiple modules may share access to the same global variables (through importing and exporting), the store holds a canonical list all global variables declared across all programs. The index k in a (**global.get** k) access refers to the k-th local variable accessible by the currently executing module. The iglobs field of the instance links this index to the position of that global variable in the store's list.

The static index parameters of all local and global variable accesses are bounds checked during type checking to ensure that they are in-bounds of their respective variable index spaces (see §2.6).

2.4.3 Memory operations

(packed types) $pt ::=$ pi8 | pi16 | pi32

(instructions) $e ::= \ldots \mid t.\textbf{load}\ (pt, sx)^?\ a\ o \mid t.\textbf{store}\ pt^?\ a\ o \mid$
memory.size | **memory.grow** | $\ldots$

(memory) $mem ::= \{\ \text{buffer} :: byte,\quad \text{max} :: nat\ \}$
(store) $S ::= \{\ \text{funcs} :: \ldots,\quad \text{tabs} :: \ldots,\quad \text{mems} :: mem^*,\quad \text{globs} :: glob^*\ \}$

These operations deal with accesses to a WebAssembly memory, which is a simple linear buffer of bytes. Each module may declare a single (possibly imported) memory, which all memory operations within the module implicitly reference through the current instance. In WebAssembly, every typed value can be linearised to a sequence of bytes of the appropriate length, and stored into a dynamically-indexed location in memory through the **store** instruction. Note that in WebAssembly, indices into memory are always of type i32. Every access to WebAssembly's linear memory is bounds-checked: both **load** and **store** will **trap** if the accessed index is out-of-bounds. Moreover, every appropriately sized sequence of bytes can be deserialised into a valid typed value through the **load** instruction; unlike in C/C++, there are no "trap representations". These operations implicitly target the *first* memory that is in-scope according to the instance. The type system guarantees that these operations are only well-typed if at least one memory is in-scope, while the language currently restricts module declarations and imports such that *at most* one memory is in-scope. When this latter restriction is relaxed, the instructions will be extended with an additional static index to allow other in-scope memories to be targetted.

$$S; F;\ (\textbf{i32.const}\ j)\ (t.\textbf{load}\ (pt, sx)^?\ a\ o)\ \hookrightarrow\ S; F; (t.\textbf{const}\ k)$$
$$\begin{aligned}\text{where}\quad & i = (F.\text{inst.imems})[0]\\ & mem = (S.\text{mems})[i].\text{buffer}\\ & ind = j + o\\ & size = |pt| \text{ or } |t| \text{ if } pt \text{ not present}\\ & |mem| \geqslant ind + size\\ & \text{from\_bytes}_{(pt,sx)^?}(\ mem[ind..ind + size)\ ) = k\end{aligned}$$

$$S; F;\ (\textbf{i32.const}\ j)\ (t.\textbf{load}\ (pt, sx)^?\ a\ o)\ \hookrightarrow\ S; F; \textbf{trap}$$
$$\begin{aligned}\text{where}\quad & i = (F.\text{inst.imems})[0]\\ & mem = (S.\text{mems})[i].\text{buffer}\\ & ind = j + o\\ & size = |pt| \text{ or } |t| \text{ if } pt \text{ not present}\\ & |mem| < ind + size\end{aligned}$$

The type-annotated $(t.\textbf{load}\ (pt, sx)^?\ a\ o)$ instruction takes a single i32 stack argument representing an offset in memory. The type annotation t determines the type of the value to be read. The optional $(pt, sx)^?$ component of the instruction (only permitted at decode-time when reading an int type) specifies that the only some number of lower-order bytes should be read from memory, with the result sign-extended (as specified by sx) to obtain the desired value. For example, the instruction (i32.**load** (pi16, u)...) would push a 4-byte i32 value onto the stack obtained by reading two lower-order bytes (pi16) and zero-extending the result. The a (alignment) component of the instruction has no semantic effect, and is purely an alignment hint to the compiler. The o (offset) component of the instruction is an additional static offset which is added to the stack argument. This allows efficient encoding of struct field accesses. Note the line $i = (F.\text{inst.imems})[0]$ which causes the instruction to load from the first (and for now, only) memory in-scope according to the instance.

$$S; F;\ (\textbf{i32.const}\ j)\ (t.\textbf{const}\ c)\ (t.\textbf{store}\ pt^?\ a\ o)\ \hookrightarrow\ S'; F; \epsilon$$
$$\begin{aligned}\text{where}\quad & i = (F.\text{inst.imems})[0]\\ & mem = (S.\text{mems})[i].\text{buffer}\\ & ind = j + o\\ & size = |pt| \text{ or } |t| \text{ if } pt \text{ not present}\\ & |mem| \geqslant ind + size\\ & S' = S \text{ with } (\text{mems}[i])[ind..ind + size) := \text{to\_bytes}_{(pt,sx)^?}(c)\end{aligned}$$

$$S; F;\ (\textbf{i32.const}\ j)\ (t.\textbf{const}\ c)\ (t.\textbf{store}\ pt^?\ a\ o)\ \hookrightarrow\ S; F; \textbf{trap}$$
$$\begin{array}{rl} \text{where} & i = (F.\text{inst.imems})[0] \\ & mem = (S.\text{mems})[i].\text{buffer} \\ & ind = j + o \\ & size = |pt| \text{ or } |t| \text{ if } pt \text{ not present} \\ & |mem| < ind + size \end{array}$$

The **store** instruction takes two stack arguments: the memory index to store into, and the value to store into that location. Its components have the same behaviour as those of **load**, although it lacks a sign-extension parameter.

$$S; F; \textbf{memory.size} \hookrightarrow S; F; (\textbf{i32.const}\ k)$$
$$\begin{array}{rl} \text{where} & i = (F.\text{inst.imems})[0] \\ & mem = (S.\text{mems})[i].\text{buffer} \\ & |mem|/2^{16} = k \end{array}$$

$$S; F; (\textbf{i32.const}\ j)\ \textbf{memory.grow} \hookrightarrow S; F; (\textbf{i32.const}\ k)$$
$$\begin{array}{rl} \text{where} & i = (F.\text{inst.imems})[0] \\ & mem = (S.\text{mems})[i].\text{buffer} \\ & |mem|/2^{16} = k \\ & k + j \leqslant (S.\text{mems})[i].\text{max} \\ & S' = S \text{ with } (\text{mems}[i]) \mathrel{+}= 0^{j*2^{16}} \end{array}$$

$$S; F; (\textbf{i32.const}\ j)\ \textbf{memory.grow} \hookrightarrow S; F; \textbf{trap}$$

A WebAssembly linear memory can be dynamically grown in size. The **memory.size** instruction returns the current size of the memory, measured in units of "pages" (each 2^{16} bytes in size). The **memory.grow** instruction takes one argument, and grows the size of memory by that many pages, up to a statically specified maximum. WebAssembly allows this instruction to non-deterministically fail, to capture the possibility of allocation failures due to memory pressures in the host environment.

While WebAssembly's AST allows an instance to hold indices for multiple memories, currently only up to one memory can be created per-module instantiation, so operations which interact with the WebAssembly memory always access the memory at the instance's first entry.

2.4.4 Control flow

$$\text{(instructions)}\quad e ::= \ldots \mid \textbf{block}\ ft\ e^*\ \textbf{end} \mid \textbf{loop}\ ft\ e^*\ \textbf{end} \mid \textbf{if}\ ft\ e^*\ \textbf{else}\ e^*\ \textbf{end} \mid \textbf{br}\ i \mid \textbf{br\_if}\ i \mid \textbf{br\_table}\ i^+ \mid \ldots$$

To explain the semantics of these instructions, we must first recall WebAssembly's concept of "administrative instructions". These are expressions which are not part of the WebAssembly program syntax, but may be produced as an intermediate step of reduction. We have already seen **trap** as an example of this. WebAssembly has two main control structures, **block** and **loop**. Their semantics are defined in terms of the **label** administrative instruction.

$$\text{(administrative instructions)}\quad e ::= \ldots \mid \textbf{label}_n\ \{e^*\}\ e^*\ \textbf{end} \mid \ldots$$

The label instruction is the basic specification building block for intra-function control flow constructions. Its behaviour is almost identical to the **label** construct proposed by Clint and Hoare [28] to model jumps out of blocks. It has three components: an *arity* n, a continuation $\{e^*\}$, and a body e^*. The body executes in the context of the label. If a **br** k instruction is executed, the k-th nested label (counting from the inside outwards in the style of de Bruijn indexing) is described as being *targetted*. Control is transferred to the continuation component of the targetted label, with the label arity determining how many stack values from the execution of the body are preserved.

Here are the formal rules for the label evaluation contexts L_k, and the related reduction rules for **label** and **br**:

$$\text{(label context)}\qquad L_0[e^*] ::= v^*\ e^*\ e'^* \qquad L_{k+1}[e^*] ::= v^*\ (\textbf{label}_n\ \{e^*_{\textsf{cont}}\}\ L_k[e^*])\ e'^*$$

$$\frac{S; F; e^* \hookrightarrow S'; F'; e'^*}{S; F; L_k[e^*] \hookrightarrow S'; F'; L_k[e'^*]}$$

$$S; F; L_0[\textbf{trap}] \hookrightarrow S; F; \textbf{trap} \qquad \text{if } L_0[\textbf{trap}] \neq \textbf{trap}$$

$$S; F; (\textbf{label}_n\ \{e^*\}\ \textbf{trap}) \hookrightarrow S; F; \textbf{trap}$$

$$S; F; (\textbf{label}_n\ \{e^*\}\ L_k[v^n\ (\textbf{br}\ k)]) \hookrightarrow S; F; v^n\ e^*$$

$$S; F; (\textbf{label}_n\ \{e^*\}\ v^*) \hookrightarrow S; F; v^*$$

A label evaluation contexts L_k represents a program fragment of k nested **label** administrative instructions, with a context "hole" in the body of the innermost label. The first reduction rule is the congruence rule for label contexts. The second and third rules describe **trap** propagating through label contexts (representing an inner error bubbling up to the top level and halting execution). The fourth rule shows the **br** instruction being

used to perform a control flow jump. As described above, the (**br** k) instruction acts as an indexed break. Counting outwards from the instruction, control is transferred to the continuation of the k-th label. Beware of off-by-one errors! Consider that a (**br** 0) instruction targets the 0-th (innermost) label. The arity of the targetted label determines the number of (stack) values which are kept and transferred to the continuation. All other values from intervening labels are discarded. The final rule represents the exiting of a label context after its body has been fully executed without breaking or trapping.

As mentioned, **block** and **loop** are defined in terms of **label**. These reduction rules are given below. In the **block** case, the continuation is empty, and the **label** arity is set to the arity of the block's output type. Therefore, when the **block** is targetted by a **br**, control jumps to the end of the block, and values are brought out of the block body to match the block's type signature.

$$v^n\ (\textbf{block}\ (t^n \to t^m)\ e^*) \hookrightarrow (\textbf{label}_m\ \{\epsilon\}\ (v^n\ e^*))$$

$$v^n\ (\textbf{loop}\ (t^n \to t^m)\ e^*) \hookrightarrow (\textbf{label}_n\ \{(\textbf{loop}\ (t^n \to t^m)\ e^*)\}\ (v^n\ e^*))$$

In the **loop** case, the continuation is the loop itself, and the **label** arity is set to the arity of the loop's *input* type. Therefore, when the **loop** is targetted by a **br**, control jumps back to the *start* of the loop, and values are brought out of the body to match the input arity of the loop. A loop will terminate when its body runs to completion without the loop being targetted by a **br**.

The **br_if** and **br_table** instructions are forms of conditional control flow. Their reduction rules are below.

$$(\textsf{i32}.\textbf{const}\ k)\ (\textbf{br\_if}\ i) \hookrightarrow (\textbf{br}\ i) \qquad \text{where } k \neq 0$$

$$(\textsf{i32}.\textbf{const}\ k)\ (\textbf{br\_if}\ i) \hookrightarrow \epsilon \qquad \text{where } k = 0$$

The **br_if** instruction takes a single i32 argument, which is interpreted as a boolean. If the argument is non-zero, the instruction acts like **br**. Otherwise, it is a no-op.

$$(t.\textbf{const}\ k)\ \textbf{br\_table}\ i^+ \hookrightarrow \textbf{br}\ (i^+[k]) \qquad \text{where } k < \text{length}(i^+)$$

$$(t.\textbf{const}\ k)\ \textbf{br\_table}\ i^+ \hookrightarrow \textbf{br}\ (\text{last}(i^+)) \qquad \text{where } k \geqslant \text{length}(i^+)$$

The **br_table** instruction carries a static non-empty list of branch targets, which is indexed by the instruction's stack argument. In the case that the index is too large, the last value of the list is used, as a default.

2.4.5 Call/return

$$\text{(instructions)}\quad e ::= \ldots \mid \textsf{call}\ i \mid \textsf{call\_indirect}\ i \mid \textsf{return}$$

$$\begin{array}{lrl}
\text{(function closure)} & cl ::= & \textsf{native}\ \{\ \text{inst} :: \mathit{inst}, \text{type} :: \mathit{ft}, \text{locals} :: t^*, \text{body} :: e^*\} \mid \\
& & \textsf{host}\ \{\ \text{type} :: \mathit{ft}, \text{host} :: \mathit{hostfunc}\} \\
\text{(table)} & \mathit{tab} ::= & \{\ \text{elems} :: i^*, \text{max} :: \mathit{nat}\} \\
\text{(store)} & S ::= & \left\{ \begin{array}{ll} \text{funcs} :: \mathit{cl}^*, & \text{tabs} :: \mathit{tab}^*, \\ \text{mems} :: \mathit{mem}^*, & \text{globs} :: \mathit{glob}^* \end{array} \right\}
\end{array}$$

$$\text{(administrative instructions)}\quad e ::= \ldots \mid \textsf{invoke}\ i \mid \textsf{frame}_n\ \{F\}\ e^*\ \textsf{end} \mid \ldots$$

$$S; F; (\textsf{call}\ k) \hookrightarrow S; F; (\textsf{invoke}\ i)$$
$$\text{where}\quad i = (F.\text{inst.ifuncs})[k]$$

Each WebAssembly module contains a list of declared and imported functions. The **call** i instruction calls the i-th function in this list. The administrative instruction **invoke** is used to give a unified semantics for function invocation; all other function call instructions are defined in terms of reduction to **invoke**. When a function is invoked, a new frame is created to execute the function body. This is represented using the **frame** administrative instruction, which keeps track of the called function's scope information, and otherwise behaves similarly to **label**.

$$S; \hat{F}; v^n\ (\textsf{invoke}\ i) \hookrightarrow S; \hat{F}; (\textsf{frame}_m\ \{F\}\ e^*)$$

$$\begin{array}{ll}
\text{where} & (S.\text{funcs})[i] = \textsf{native}\ \{\text{inst}\ \mathit{inst}, \text{type}\ t^n \to t^m, \text{locals}\ t^p, \text{body}\ e^*\} \\
& F = \{\text{locs}\ (v^n)(0^p), \text{inst}\ \mathit{inst}\}
\end{array}$$

Note that when a function is invoked, its arguments become available as local variables within the function body, along with its explicitly declared local variables which are zero-initialised.

The **return** instruction transfers control to the end of the function, breaking out of the **frame** in the same way that **br** breaks from a **label**. Reduction rules for **frame** and **return**:

$$\frac{S; F; e^* \hookrightarrow S'; F'; e'^*}{S; \hat{F}; (\textsf{frame}_n\ \{F\}\ e^*) \hookrightarrow S'; \hat{F}; (\textsf{frame}_n\ \{F'\}\ e'^*)}$$

$$S; \hat{F}; (\textsf{frame}_n\ \{F\}\ \textsf{trap}) \hookrightarrow S; \hat{F}; \textsf{trap}$$

$$S; \hat{F}; (\textsf{frame}_n\ \{F\}\ v^*) \hookrightarrow S; \hat{F}; v^*$$

$$S; \hat{F}; (\textsf{frame}_n\ \{F\}\ L_k[v^n\ \textsf{return}]) \hookrightarrow S; \hat{F}; v^n$$

WebAssembly modules are also allowed to declare and import a mutable table of function references. The **call_indirect** instruction dynamically calls an indexed function in the table. It carries an index which references a type annotation which must be matched at runtime, else a **trap** occurs. This is WebAssembly's mechanism for dynamic dispatch.

$$S; F; (\textbf{i32.const}\ k)\ (\textbf{call\_indirect}\ j) \hookrightarrow S; F; (\textbf{invoke}\ i)$$
$$\text{where}\quad i_t = (F.\text{inst.itabs})[0]$$
$$tab = S.\text{tabs}[i_t]$$
$$i = tab[k]$$
$$(S.\text{funcs})[i].\text{type} = S.\text{types}[j]$$

$$S; F; (\textbf{i32.const}\ k)\ (\textbf{call\_indirect}\ j) \hookrightarrow S; F; \textbf{trap}$$
$$\text{otherwise}$$

WebAssembly currently offers no instructions to mutate the table. This capability will be introduced in a future feature [25]. However, the WebAssembly program may also call *host functions*. These functions are provided by the environment, and their semantics are outside the scope of WebAssembly's specification. Among other things, a host function may be used to mutate a program's table, if it is exported. On the Web, the canonical host is JavaScript.

To specify host functions, WebAssembly simply axiomatises the existence of a relation describing the behaviour of the host. When a host function is executed, this relation explicitly describes whether the function runs forever or terminates.

$$S; \hat{F}; v^n\ (\textbf{invoke}\ i) \hookrightarrow S'; \hat{F}; v'^m$$
$$\text{where}\quad (S.\text{funcs})[i] = \mathsf{host}\ \{\text{type}\ t^n \to t^m, \text{host}\ hostfunc\}$$
$$hostfunc(S, t^m, v^n) \ni (S', v'^m)$$

$$S; \hat{F}; v^n\ (\textbf{invoke}\ i) \hookrightarrow S; \hat{F}; v^n\ (\textbf{invoke}\ i)$$
$$\text{where}\quad (S.\text{funcs})[i] = \mathsf{host}\ \{\text{type}\ t^n \to t^m, \text{host}\ hostfunc\}$$
$$hostfunc(S, t^m, v^n) \ni \bot$$

The execution of the host function is subject to some axiomatic constraints which ensure that the integrity of the WebAssembly state is preserved.

$$hostfunc(S, t^m, v^n) \ni (S', v^m) \implies t^m = \text{map typeof}\ v^m \wedge$$
$$S \prec_{\mathsf{s}} S' \wedge$$
$$(\vdash_{\mathsf{s}} S : \mathsf{ok} \implies \vdash_{\mathsf{s}} S' : \mathsf{ok})$$

These relations are defined in §2.7. The conditions on *hostfunc* essentially amount to an axiomatisation of the minimal guarantees that all instructions must satisfy to ensure type soundness. As discussed in §3.1.4, in the course of mechanising the specification, some issues with an earlier version of this condition were found and corrected.

2.5 WebAssembly modules

WebAssembly programs are organised using *modules*. A module consists of a collection of declarations for global state objects: global variables, tables, memories, and functions, together with some initialization information. Before the code of a module can be executed, the module must be *instantiated*. This is a linking and allocation phase which inserts objects corresponding to the module's declarations into the current global store S. Parts of the module state can be declared as "imports". At instantiation time, these imports must be satisfied by objects of the appropriate type which were allocated by previous instantiations. The formal structure of a module is given in Fig. 2.1.

Before a module can be instantiated, it must be *validated*. This is a type checking procedure which ensures that the module is well-formed, and that the code of its functions obeys certain restrictions. We will first describe the type checking procedure for instruction sequences, and the associated type soundness theorem, before describing the top-level lifecycle of the WebAssembly module (§2.8).

$$
\begin{array}{lrl}
\text{(limits)} & \mathit{limit}, \mathit{mt}, \mathit{tt} & ::= \{\text{min} :: i, \text{max} :: i\} \\
\text{(global type)} & \mathit{gt} & ::= \{\text{mut} :: \mathit{mut}, \text{type} :: t\} \\
\text{(functions)} & \mathit{funcdecl} & ::= \mathsf{func}\ i\ (\mathsf{local}\ t^*)\ e^* \\
\text{(globals)} & \mathit{globdecl} & ::= \mathsf{global}\ \mathit{gt}\ e^* \\
\text{(tables)} & \mathit{tabdecl} & ::= \mathsf{table}\ \mathit{tt} \\
\text{(memories)} & \mathit{memdecl} & ::= \mathsf{memory}\ \mathit{mt} \\
\text{(elem segments)} & \mathit{elemdecl} & ::= \{\text{addr} :: i, \text{off} :: e^*, \text{init} :: i^*\} \\
\text{(data segments)} & \mathit{datadecl} & ::= \{\text{addr} :: i, \text{off} :: e^*, \text{init} :: \mathit{byte}^*\} \\
\text{(import descriptor)} & \mathit{idesc} & ::= \mathsf{ifunc}\ i \mid \mathsf{imem}\ \mathit{mt} \mid \mathsf{itab}\ \mathit{tt} \mid \mathsf{iglob}\ \mathit{gt} \\
\text{(imports)} & \mathit{imp} & ::= \mathsf{import} \left\{ \begin{array}{l} \text{namespace} :: \mathit{string}, \\ \text{name} :: \mathit{string}, \\ \text{desc} :: \mathit{idesc} \end{array} \right\} \\
\text{(export descriptor)} & \mathit{edesc} & ::= \mathsf{efunc}\ i \mid \mathsf{emem}\ i \mid \mathsf{etab}\ i \mid \mathsf{eglob}\ i \\
\text{(exports)} & \mathit{exp} & ::= \mathsf{export}\ \{\text{name} :: \mathit{string}, \text{desc} :: \mathit{edesc}\} \\
\text{(modules)} & \mathit{module} & ::= \left\{ \begin{array}{l} \text{types} :: \mathit{ft}^*,\ \text{funcs} :: \mathit{funcdecl}^*,\ \text{globs} :: \mathit{globdecl}^*, \\ \text{tabs} :: \mathit{tabdecl}^*,\ \text{mems} :: \mathit{memdecl}^*, \\ \text{elem} :: \mathit{elemdecl}^*,\ \text{data} :: \mathit{datadecl}^*, \\ \text{start} :: i,\ \text{imports} :: \mathit{imp}^*,\ \mathit{exports} :: \mathit{exp}^* \end{array} \right\}
\end{array}
$$

Figure 2.1: WebAssembly module abstract syntax.

2.6 Type System

Some of the WebAssembly instructions described previously have their reduction rules defined only when the state is a certain shape. For example, a *t.binop* instruction can only execute if two values of type t lie at the top of the stack. All WebAssembly programs undergo *validation* (type checking) to ensure that attempting to execute such an instruction will never result in a stuck state (i.e. that the shape of the stack always satisfies the necessary conditions for reduction). The guarantees of the type system are strong enough that implementations do not need to simulate/check the shape of the stack while executing, and can instead compile operations on the stack to a Single Static Assignment (SSA) form on registers [29]. At a high level, the type system needs to check the following things:

- An instruction is only allowed at a certain program point if there are guaranteed to be the necessary consumable values on the top of the stack (with the appropriate types) for it to execute.
- When an instruction uses a static index to reference part of the state (e.g. local and global variables, or the **call** instruction), that index must be in bounds. The type of the instruction is dependent on the type of the referenced state (e.g. **local.get** i has type $\epsilon \to t$, where t is the statically-declared type of the local variable indexed by i).
- A (**br** k) instruction is only allowed in a context with at least k enclosing labels, and there must be enough values on the stack to satisfy the required type associated with that label (see Fig. 2.6).

Each WebAssembly function is explicitly type-annotated with the expected type of its body. Similarly, every control flow join (e.g. **block**, **loop**) is explicitly type-annotated. These type annotations mean that full type inference is unnecessary: when typing the body of a function or block, the expected input and output types are always known ahead of time. Typing of a code fragment in WebAssembly uses a typing context C (see Fig. 2.2) to represent the global, function-local, and block-local type annotations which are currently in scope. This is determined statically by the declarations of the enclosing module, function, and evaluation context. The func, global, table, and memory components of the typing context hold the types of the globally scoped object declarations of the module. We will formally describe the details of how C is related to the module definition in §2.8. The local component holds the types of the function-scoped local variables. The label and return components hold the target types of the **br**-targetted (**block** and **loop**) and **return**-targetted (**func**) contexts respectively.

Typing context

$$\text{(contexts)}\quad C \;::=\; \left\{ \begin{array}{l} \text{type} :: ft^*, \text{func} :: ft^*,\ \text{table} :: tt^*,\ \text{memory} :: mt^*,\ \text{global} :: gt^*, \\ \text{local} :: t^*,\ \text{label} :: (t^*)^*,\ \text{return} :: (t^*)^? \end{array} \right\}$$

Instruction sequences typing judgement

$C \vdash e^* : ft$

Figure 2.2: Typing context and top-level shape of instruction sequences typing.

We will now describe the typing rules for each individual WebAssembly instruction.

$$\frac{}{C \vdash t.\mathsf{const}\ c : \epsilon \to t} \qquad \frac{}{C \vdash t.unop_t : t \to t} \qquad \frac{}{C \vdash t.binop_t : t\ t \to t}$$

$$\frac{}{C \vdash t.testop : t \to \mathsf{i32}} \qquad \frac{}{C \vdash t.relop_t : t\ t \to \mathsf{i32}} \qquad \frac{}{C \vdash t_2.cvtop_{(t_1,sx^?)} : t_1 \to t_2}$$

$$\frac{}{C \vdash \mathsf{nop} : \epsilon \to \epsilon} \quad \frac{}{C \vdash \mathsf{drop} : t \to \epsilon} \quad \frac{}{C \vdash \mathsf{select} : t\ t\ \mathsf{i32} \to t}$$

$$\frac{}{C \vdash \mathsf{unreachable} : t_1^* \to t_2^*} \qquad \frac{}{C \vdash \epsilon : \epsilon \to \epsilon}$$

$$\frac{C \vdash e_1^* : t_1^* \to t_2^* \quad C \vdash e_2 : t_2^* \to t_3^*}{C \vdash e_1^*\ e_2 : t_1^* \to t_3^*}\ [comp] \qquad \frac{C \vdash e^* : t_1^* \to t_2^*}{C \vdash e^* : t^*\ t_1^* \to t^*\ t_2^*}\ [weaken]$$

Figure 2.3: Basic typing.

The rules of Fig. 2.3 deal with instructions that only interact with WebAssembly's stack. The instruction typing rules have no premises, and simply ensure that the correct types are produced and consumed by straight-line code. The composition rule allows types to be sequentially composed in the intuitive way, while the weakening rule allows types to be extended, representing an untouched "stack base".

$$\frac{C.\text{local}[i] = t}{C \vdash \mathsf{local.get}\ i : \epsilon \to t} \qquad \frac{C.\text{local}[i] = t}{C \vdash \mathsf{local.set}\ i : t \to \epsilon} \qquad \frac{C.\text{local}[i] = t}{C \vdash \mathsf{local.tee}\ i : t \to t}$$

$$\frac{C.\text{global}[i].\text{type} = t}{C \vdash \mathsf{global.get}\ i : \epsilon \to t} \qquad \frac{C.\text{global}[i] = \{\text{mut}\ \mathsf{mut}, \text{type}\ t\}}{C \vdash \mathsf{global.set}\ i : t \to \epsilon}$$

Figure 2.4: Context typing 1.

The rules of Fig. 2.4 check that statically-indexed variable accesses are in-bounds. Recall that in the metatheory of the WebAssembly specification, array accesses in premises must be in-bounds for a derivation to succeed. The global component of the context represents the top-level module global variable declarations, while the local component of the context represents the function-scope argument and scratch (zero-initialised) local variable declarations (see §2.4.5).

$$\frac{C.\text{memory}[0] = mt \qquad 2^a \leqslant |t|}{C \vdash t.\textbf{load} \ - \ a \ o : \text{i32} \to t} \qquad \frac{C.\text{memory}[0] = mt \qquad 2^a \leqslant |pt| < |t| \qquad t = \text{i}n}{C \vdash t.\textbf{load} \ (pt, sx) \ a \ o : \text{i32} \to t}$$

$$\frac{C.\text{memory}[0] = mt \qquad 2^a \leqslant |t|}{C \vdash t.\textbf{store} \ - \ a \ o : \text{i32} \ t \to \epsilon} \qquad \frac{C.\text{memory}[0] = mt \qquad 2^a \leqslant |pt| < |t| \qquad t = \text{i}m}{C \vdash t.\textbf{store} \ pt \ a \ o : \text{i32} \ t \to \epsilon}$$

$$\frac{C.\text{memory}[0] = mt}{C \vdash \textbf{memory.size} : \epsilon \to \text{i32}} \qquad \frac{C.\text{memory}[0] = mt}{C \vdash \textbf{memory.grow} : \text{i32} \to \text{i32}}$$

Figure 2.5: Context typing 2.

The instructions of Fig 2.5 are only allowed to occur in the program if the module has declared or imported a memory. Currently, a module may only have a single memory in scope (which may have been imported from another module), which all of these instructions implicitly index. However, the specification is designed with the expectation that a module may include multiple memories in future, which will require extended instructions to access. The checks on the alignment hint a in the rules for **load** and **store** simply ensure that an instruction cannot be hinted to have an alignment greater than the natural alignment of its type annotation.

$$\frac{ft = t_1^n \to t_2^m \qquad \{\text{label } t_2^m\} \oplus C \vdash e^* : ft}{C \vdash \textbf{block} \ ft \ e^* \ \textbf{end} : ft} \qquad \frac{ft = t_1^n \to t_2^m \qquad \{\text{label } t_1^n\} \oplus C \vdash e^* : ft}{C \vdash \textbf{loop} \ ft \ e^* \ \textbf{end} : ft}$$

$$\frac{ft = t_1^n \to t_2^m \qquad \{\text{label } t_2^m\} \oplus C \vdash e_1^* : ft \qquad \{\text{label } t_2^m\} \oplus C \vdash e_2^* : ft}{C \vdash \textbf{if} \ ft \ e_1^* \ \textbf{else} \ e_2^* \ \textbf{end} : t_1^n \ \text{i32} \to t_2^m}$$

$$\frac{C.\text{label}[i] = t^*}{C \vdash \textbf{br} \ i : t_1^* \ t^* \to t_2^*} \qquad \frac{C.\text{label}[i] = t^*}{C \vdash \textbf{br\_if} \ i : t^* \ \text{i32} \to t^*} \qquad \frac{(C.\text{label}[i] = t^*)^+}{C \vdash \textbf{br\_table} \ i^+ : t_1^* \ t^* \ \text{i32} \to t_2^*}$$

Figure 2.6: Control typing 1.

The typing rules of Fig. 2.6 ensure that **br** instructions have valid indices, and that the stack will be in the correct state when they execute. Recall that **br** transfers control to either the *end* of a **block**, or the *start* of a **loop**. When control is transferred by **br** to the end of a **block**, the top of the stack must match the *output* type of the block. Similarly, when control is transferred to the start of a **loop**, the top of the stack must match the *input* type of the loop.

This is enforced by the label component of C. The body of a **block** or **loop** must be typed to match the construct's type annotation. The **if** instruction acts like a block, and *both* possibly executed bodies must have the same type. While typing the body, the context C is extended with a *label* which keeps track of the required type if a **br** is

executed which targets that construct. The rules for various flavours of **br** then check this context to determine what their input type should be. Note a syntactic convention used in the typing rule for **br_table**, where the use of + as a superscript in the premise indicates that the premise is repeated over every element of i^+ (with the i of $C.\text{label}[i]$ in the premise binding to each element). This convention is common in future rules.

There are subtle differences in the types of each **br**. When executing **br** or **br_table**, any sequentially subsequent code is syntactically dead. However, it is a (contentious) design decision of WebAssembly that even dead code should be well-typed (see §3.3.1). An arbitrary type can be picked as the output type of **br** and **br_table**, but it is still possible for subsequent code to be ill-typed. The rationale behind this will be discussed in more detail in §3.3.1. The **br_if** instruction is conditional, so it is possible for execution to continue without a control transfer occurring. Therefore, subsequent code must be typed as though the stack is unchanged, aside from the i32 argument consumed by **br_if**.

$$\frac{C.\text{return} = t^*}{C \vdash \textbf{return} : t_1^*\ t^* \to t_2^*} \qquad \frac{C.\text{func}[i] = ft}{C \vdash \textbf{call}\ i : ft} \qquad \frac{C.\text{type}[i] = t_1^* \to t_2^* \qquad C.\text{table}[0] = tt}{C \vdash \textbf{call\_indirect}\ i : t_1^*\ \text{i32} \to t_2^*}$$

Figure 2.7: Control typing 2.

Finally, the rules of Fig. 2.7 deal with function calls. The return component of C is only set at the top level when beginning to type a function. Because all functions are typed to obey their explicit type annotation, the type of **call** is simply the type annotation of the statically indexed function being called. The **call_indirect** instruction, which is only well-typed if the module declares or imports a table, is given an explicit type annotation to use as its type (that must be dynamically checked during execution), using the same index space as the function type annotations.

2.7 The type soundness statement

The correctness condition of WebAssembly's type system is a textbook example of *syntactic type soundness* [30]. The official specification states the desired type soundness property, but does not provide a full proof. Syntactic type soundness consists of two properties:

Preservation If a WebAssembly configuration is well-typed, then if it reduces (executes) one step, the resulting configuration will be well-typed.

Progress If a WebAssembly configuration is well-typed, then either it represents a terminated program (in the sense that it has reduced to a bare list of values or **trap**), or there exists a reduction step it is allowed to make.

In order to state these properties, WebAssembly's type system must be augmented to fully type the language's runtime configurations, including the runtime-only administrative instructions. This is done by extending the typing judgement to be parameterised by the store, as some details of the types of the administrative instructions depend on values in the store. For example, the type of (**invoke** i) depends on the type of the function in the store that i indexes. As noted by Pierce [31], this extension of the typing relation with a store parameter is a well-known naive approach to formulating type soundness, and is often not sufficient for a real language since locations in the store may change their types dynamically at runtime or contain cyclical references. However, because every object in the WebAssembly store is associated with a statically known type (through its associated module declaration), this naive approach is sufficient.

The judgement $\vdash_{\mathsf{c}} S; F; e^* : t^*$ can be intuitively understood as typing a top-level WebAssembly configuration which can execute without any additional context needed, and if the configuration terminates without error it will produce a list of values of type t^*. The judgement is formalised in an appendix of the official WebAssembly specification [32], and is defined as follows:

Table and memory validity These judgements associate a (possibly imprecise) *limit type* with a given runtime table/memory. It is an invariant of the semantics that the limit type of a memory will stay valid even if the memory grows in size.

$$\frac{min \geqslant min' \qquad (min \leqslant max)^? \qquad (max \leqslant max')^?}{\vdash_{\mathsf{limit}} \{\text{min } min, \text{max } max^?\} : \{\text{min } min', \text{max } max'^?\}}$$

$$\frac{\vdash_{\mathsf{limit}} \{\text{min } |t.\text{elems}|, \text{max } t.\text{max}\} : lim}{\vdash_{\mathsf{tab}} t : lim}$$

$$\frac{\vdash_{\mathsf{limit}} \{\text{min } |m.\text{buffer}|/2^{16}, \text{max } m.\text{max}\} : lim}{\vdash_{\mathsf{mem}} m : lim}$$

Global validity This judgement associates a global variable with its global type.

$$\frac{g.\text{mut} = gt.\text{mut} \qquad \text{typeof}(g.\text{val}) = gt.\text{type}}{\vdash_{\mathsf{glob}} g : gt}$$

Instance validity This judgement associates a runtime *instance*, containing references to objects in the global store S, with a type context C which has the same structure but contains the *types* of each object instead.

$$\frac{\begin{array}{c}(S.\text{funcs}[if].\text{type} = tf)^p \quad (\vdash_{\mathsf{tab}} (S.\text{tabs}[it]) : tt)^n \\ (\vdash_{\mathsf{mem}} (S.\text{mems}[im]) : tm)^m \quad (\vdash_{\mathsf{glob}} (S.\text{globs}[ig]) : tg)^l \\ inst = \{\text{types } ft^q, \text{ifuncs } if^p, \text{itabs } it^n, \text{imems } im^m, \text{iglobs } ig^l\} \\ C = \{\text{type } ft^q, \text{func } tf^p, \text{table } tm^m, \text{memory } tg^l, \text{global } tg^n\}\end{array}}{S \vdash_{\mathsf{i}} inst : C}$$

Administrative instruction validity This judgement is defined as a "generalisation" of the instruction typing judgement $C \vdash e^* : ft$, adding an additional store component S (a standard approach described by Pierce [31]). The premise of **frame** refers to the local validity judgement below, which itself depends on administrative instruction validity, so these definitions are mutually recursive.

$$\frac{}{S; C \vdash \mathbf{trap} : ft}$$

$$\frac{S; C \vdash e_l^* : t^n \to t_l^* \qquad S; \{\text{label } t^n\} \oplus C \vdash e^* : \epsilon \to t_l^*}{S; C \vdash \mathbf{label}_n\ \{e_l^*\}\ e^* : \epsilon \to t_l^*}$$

$$\frac{(S.\text{funcs}[i]).\text{type} = ft}{S; C \vdash \mathbf{invoke}\ i : ft}$$

$$\frac{S; (t^n) \vdash_{\mathsf{loc}} F; e^* : t^*}{S; C \vdash \mathbf{frame}_n\ \{F\}\ e^* : \epsilon \to t^*}$$

Frame validity This judgement lifts instance validity to frames, additionally populating the type context C with the types of the frame's local variables.

$$\frac{(\ t_v = \text{typeof}(v)\ )^n \qquad F.\text{locs} = v^n \qquad S \vdash_{\mathsf{i}} F.\text{inst} : C}{S \vdash_{\mathsf{f}} F : C[\text{local}\ :=\ t_v^n]}$$

Local validity This judgement defines the typing of an instruction sequence under a given frame. The typing context under which the instruction sequence is typed is determined by the frame validity judgement above. Note that this definition is mutually recursive with the definition of administrative instruction validity.

$$\frac{S \vdash_{\mathsf{f}} F : C \qquad S; C[\text{return} := (t_r^*)^?] \vdash e^* : \epsilon \to t^*}{S; (t_r^*)^? \vdash_{\mathsf{loc}} F; e^* : t^*}$$

Function closure validity A native function closure consists of an instruction sequence together with an associated instance, local type declarations, and type signature. This judgement lifts validity of instruction sequences to validity of function closures. Note that it is guaranteed by construction that the function body will contain no administrative instructions.

$$\frac{S \vdash_{\mathsf{i}} inst : C \qquad C[\text{locals} := t^* t_l^*, \text{labels} := t'^*, \text{return} := t'^*] \vdash e^* : \epsilon \to t'^*}{S \vdash_{\mathsf{cl}} \mathsf{native}\{\text{inst} :: inst, \text{type} :: t^* \to t'^*, \text{locals} :: t_l^*, \text{body} :: e^*\} : t^* \to t'^*}$$

$$\frac{}{S \vdash_{\mathsf{cl}} \mathsf{host}\{\text{type} :: ft, \text{host} :: hostfunc\} : ft}$$

Store validity This judgement checks that all components of the store are well-formed.

$$\frac{(S \vdash_{\mathsf{cl}} cl : ft)^* \quad (\vdash_{\mathsf{tab}} t : tt)^* \quad (\vdash_{\mathsf{mem}} m : mt)^* \qquad S = \{\text{funcs } cl^*, \text{tabs } t^*, \text{mems } m^*, \text{globs } g^*\}}{\vdash_{\mathsf{s}} S : \mathsf{ok}}$$

Configuration validity This is the top-level judgement used to define type soundness.

$$\frac{\vdash_{\mathsf{s}} S : \mathsf{ok} \quad S; \epsilon \vdash_{\mathsf{loc}} F; e^* : t^*}{\vdash_{\mathsf{c}} S; F; e^* : t^*}$$

Store extension This judgement defines a relation which expresses a bound on how the store is allowed to change over time. In particular, the types of existing components of the store may never change (although more components may be appended).

Store extension does not inherently preserve store validity, since it does not describe whether newly-appended store components are well-formed, but it does preserve the well-formedness of existing components. Similarly, preservation of store validity alone does not imply store extension.

$$\frac{|t.\text{elems}| \leqslant |t'.\text{elems}| \qquad t.\text{max} = t'.\text{max}}{t \prec_{\mathsf{tab}} t'}$$

$$\frac{|m.\text{buffer}| \leqslant |m'.\text{buffer}| \qquad m.\text{max} = m'.\text{max}}{m \prec_{\mathsf{mem}} m'}$$

$$\frac{g.\text{mut} = g'.\text{mut} \qquad \text{typeof}(g.\text{val}) = \text{typeof}(g'.\text{val}) \qquad g.\text{mut} \implies g.\text{val} = g'.\text{val}}{g \prec_{\mathsf{glob}} g'}$$

$$\frac{(t \prec_{\mathsf{tab}} t')^b \qquad (m \prec_{\mathsf{mem}} m')^c \qquad (g \prec_{\mathsf{glob}} g')^d \qquad S = \{\text{funcs } cl^a, \text{tabs } t^b, \text{mems } m^c, \text{globs } g^d\} \qquad S' = \{\text{funcs } cl^a cl''^*, \text{tabs } t'^b t''^*, \text{mems } m'^c m''^*, \text{globs } g'^d g''^*\}}{S \prec_{\mathsf{s}} S'}$$

Now, the top-level progress and preservation properties can be fully formalised:

Progress If $\vdash_{\mathsf{c}} S; F; e^* : t^*$, then
$$e^* = [\mathbf{trap}] \vee (\exists v^*.\ e^* = v^*) \vee \exists S' F' e'.\ S; F; e^* \hookrightarrow S'; F'; e'^*.$$

This property states that a well-typed configuration that is not *terminal* (a bare list of values, or an exceptional **trap**) will always be able to reduce a further step. Among other things, this implies that operations which pop from the value stack are guaranteed by the type system to only execute if enough values are on the stack for the pop to be successful.

Preservation If $\vdash_{\mathsf{c}} S; F; e^* : t^*$ and $S; F; e^* \hookrightarrow S'; F'; e'^*$, then
$$\vdash_{\mathsf{c}} S'; F'; e'^* : t^* \text{ and } S \prec_{\mathsf{s}} S'$$

This property states that if a well-typed WebAssembly configuration reduces one step, the resulting reduct is typeable with the same type, and respects store extension.

2.8 WebAssembly module lifecycle

As previously mentioned, WebAssembly code is distributed and compiled as one or more "modules". Each module represents a single WebAssembly compilation unit. At a high-level, WebAssembly code goes through a number of distinct phases in order: decoding, validation, compilation, instantiation, and execution.

1. **Decoding.** The WebAssembly module, as a stream of bytes, is decoded/parsed into the structured definition of Fig. 2.1. Decoding of WebAssembly is formalised in its own section of the official specification, but we will not deal with this step explicitly in any of the work described in this thesis.

2. **Validation.** The declarations and code of the decoded module are checked for type safety. As one part of this, the body of each declared function is checked according to the rules defined in §2.6. The formal details of top-level module validation will be described below (§2.8.1).

3. **Compilation.** The code of the module is compiled to platform machine code. No modification of the WebAssembly abstract state takes place in this phase and exported functions cannot yet be accessed or called. The definitions of decoding and validation are arranged so that an efficient implementation can perform decoding, validation, and compilation all at once on a partially downloaded module definition, in a streaming fashion.

4. **Instantiation.** A runtime *instance* of the WebAssembly module is created. This involves the allocation of function closures, global variables, tables, memory etc, as defined in the declarations of the module. References to existing allocations must

be provided to satisfy the module's imports. The new instance holds references to all created and imported state. A module may be instantiated multiple times, and separate global state will be created for each instance. For example, a module declaring a memory, instantiated twice, will allocate separate new memories for each instance to reference. The compiled code of the previous step usually contains indirections so that it may be closed over by the current instance, allowing multiple instances to share the same compiled code (without requiring recompilation) but access different state.

5. **Execution.** The instance allows access to exported definitions. An exported function closure (containing a reference to the instance, and the compiled code associated with it) may be executed, according to the semantics described in §2.4.

The most esoteric aspect of the module lifecycle is the *instantiation* stage. The intention of this arrangement is that compilation is performed once per module, and is relatively expensive, but can be performed (along with decoding and validation) in a streaming manner as the module is still being downloaded. While a module may import additional state from the environment, or other modules, a well-typed module contains enough type information to be compiled *before any imports are satisfied*. Instantiation is then a relatively cheap operation which satisfies imports and creates the necessary runtime state for the code to execute. It is expected that a compiled module may be instantiated multiple times, and therefore a typical WebAssembly implementation will compile the module to be parameterised over a given instance, meaning no additional code generation is necessary at the instantiation step. This does mean that memory accesses contain an additional indirection: the compiled code must first retrieve the memory address from the instance before applying the given offset. Surprisingly, this additional indirection apparently caused minimal performance loss, reported as less than 1% in V8 [33].

The formal details of instantiation can be found in [34] but do not need to be discussed here in detail. While the definition of instantiation is mechanised as part of the implementation of the verified interpreter described in §3.3.2, no proofs are performed against it as part of this thesis. At a high level, instantiation is defined for a module together with a list of imports. The provided imports are type checked to ensure they match the import declarations of the module. If successful, each declaration of the provided module is allocated and appended to the global store S. The instance *inst* used during reduction is the collection of references to these allocations.

2.8.1 Validation formally

Module validation, and auxiliary definitions, are defined below. Most definitions are concerned with building the context C in order to validate each function body.

Declarations

These relations assign types to declarations of global state made in the top level of the module (see §2.1).

$$\frac{ft = t_1^* \to t_2^* \qquad C[\text{locals} := t_1^* t_l^*, \text{labels} := t_2^*, \text{return} := t_2^*] \vdash e^* : \epsilon \to t_2^*}{C \vdash_{\mathsf{func_m}} \textbf{func}\ ft\ (\textbf{local}\ t_l^*)\ e^* : ft}$$

$$\frac{gt = \mathsf{mut}^?\ t \qquad C \vdash_{\mathsf{const}} e^* : \epsilon \to t}{C \vdash_{\mathsf{glob_m}} \textbf{global}\ gt\ e^* : gt}$$

$$\frac{tt = \{\text{min}\ min, \text{max}\ max^?\} \qquad min \leqslant 2^{32} \qquad (max \leqslant 2^{32} \qquad min \leqslant max)^?}{\vdash_{\mathsf{tab_m}} \textbf{table}\ tt : tt}$$

$$\frac{mt = \{\text{min}\ min, \text{max}\ max^?\} \qquad min \leqslant 2^{16} \qquad (max \leqslant 2^{16} \qquad min \leqslant max)^?}{\vdash_{\mathsf{mem_m}} \textbf{memory}\ mt : mt}$$

Initialisation segments

These relations check the validity of *segment* declarations, which declare initial values with which to populate the module's memory and table.

$$\frac{C.\text{memory}[i_m] = mt \qquad C \vdash_{\mathsf{const}} e^* : \epsilon \to \mathsf{i32} \qquad (C.\text{func}[i] = ft)^*}{C \vdash_{\mathsf{elem_m}} \{\text{addr}\ i_m, \text{off}\ e^*, \text{init}\ i^*\} : \mathsf{ok}}$$

$$\frac{C.\text{table}[i_t] = tt \qquad C \vdash_{\mathsf{const}} e^* : \epsilon \to \mathsf{i32}}{C \vdash_{\mathsf{data_t}} \{\text{addr}\ i_t, \text{off}\ e^*, \text{init}\ byte^*\} : \mathsf{ok}}$$

Imports and exports

These definitions associate module imports and exports with types, allowing the module to be given a type signature in terms of the imports it expects and the exports it produces.

$$\text{(external types)}\quad ext ::= \mathsf{func_t}\ ft \mid \mathsf{mem_t}\ mt \mid \mathsf{tab_t}\ tt \mid \mathsf{glob_t}\ gt$$

$$\frac{C.\text{type}[i] = ft}{C \vdash_{\mathsf{imp}} \mathsf{ifunc}\ i : \mathsf{func_t}\ ft} \qquad \frac{}{C \vdash_{\mathsf{imp}} \mathsf{imem}\ mt : \mathsf{mem_t}\ mt}$$

$$\frac{}{C \vdash_{\mathsf{imp}} \mathsf{itab}\ tt : \mathsf{tab_t}\ tt} \qquad \frac{}{C \vdash_{\mathsf{imp}} \mathsf{iglob}\ gt : \mathsf{glob_t}\ gt}$$

$$\frac{C.\text{func}[i] = ft}{C \vdash_{\mathsf{exp}} \mathsf{efunc}\ i : \mathsf{func_t}\ ft} \qquad \frac{C.\text{memory}[i] = mt}{C \vdash_{\mathsf{exp}} \mathsf{emem}\ i : \mathsf{mem_t}\ mt}$$

$$\frac{C.\text{table}[i] = tt}{C \vdash_{\mathsf{exp}} \mathsf{etab}\ i : \mathsf{tab_t}\ tt} \qquad \frac{C.\text{global}[i] = gt}{C \vdash_{\mathsf{exp}} \mathsf{eglob}\ i : \mathsf{glob_t}\ gt}$$

Module

$$
\begin{array}{ll}
① & (C \vdash_{\mathsf{func_m}} fdecl : ft)^* \quad (C' \vdash_{\mathsf{glob_m}} gdecl : gt)^* \quad (\vdash_{\mathsf{tab_m}} tdecl : tt)^* \quad (\vdash_{\mathsf{mem_m}} mdecl : mt)^* \\
② & (C \vdash_{\mathsf{elem_m}} edecl : \mathsf{ok})^* \quad (C \vdash_{\mathsf{data_m}} ddecl : \mathsf{ok})^* \quad i_s < \mathrm{length}(C.\mathrm{func}) \\
③ & (C \vdash_{\mathsf{imp}} imp : impt)^* \quad (C \vdash_{\mathsf{exp}} exp : expt)^* \quad (exp.\mathrm{name})^* \text{ distinct} \\
④ & ft^*_{imp} = \mathrm{funcs}(imp^*) \quad gt^*_{imp} = \mathrm{globs}(impt^*) \quad tt^*_{imp} = \mathrm{tabs}(impt^*) \quad mt^*_{imp} = \mathrm{mems}(impt^*) \\
⑤ & C = \left\{ \begin{array}{l} \text{type } tf^*, \text{func } ft^*_{imp}\, ft^*,\ \text{global } gt^*_{imp}\, gt^*,\ \text{table } tt^*_{imp}\, tt^*,\ \text{memory } mt^*_{imp}\, mt^*, \\ \text{local } \epsilon,\ \text{label } \epsilon,\ \text{return } \epsilon \end{array} \right\} \\
⑥ & C' = \{\text{type } \epsilon, \text{func } \epsilon, \text{global } gt^*_{imp}, \text{table } \epsilon, \text{memory } \epsilon, \text{local } \epsilon, \text{label } \epsilon, \text{return } \epsilon\}
\end{array}
$$

$$
\vdash_{\mathsf{module}} \left\{ \begin{array}{l} \text{types } tf^*, \text{funcs } fdecl^*,\ \text{globs } gdecl^*,\ \text{tabs } tdecl^*,\ \text{mems } mdecl^*, \\ \text{elem } edecl^*,\ \text{data } ddecl^*,\ \text{start } i_s,\ \text{imports } imp^*,\ \textit{exports } exp^* \end{array} \right\} : impt^* \to expt^*
$$

Line ① of the premises in the module validation rule describes the four main globally declared module components (functions, globals, tables, and memories) being checked for well-formedness and given a type which is later used to define the corresponding component of the type context C. Line ② shows the element and data segments (initialising the table and memory respectively) being checked for well-formedness. The start function index is also checked to ensure it lies within the range of imported/declared functions. Line ③ shows the modules' declared imports and exports being associated with types. Line ④ shows the imports being filtered according to their type in order to build to type context C. Line ⑤ shows the import types and the types of declared module components are used to define C. By convention, imported definitions appear first in the index space of each component, followed by the declarations of the module itself. The use of the partial context C' in line ⑥ is to ensure that the initial values of global variables may not refer to each other circularly. Note that the definition given for C is circular (but still inductive), since the function types derived from validating the function bodies in the first line are themselves components of C. This is merely a stylistic choice of the specification; real implementations break the circularity by building C from just the function type annotation before validating the body to check that the annotation matches.

Chapter 3

Soundness Proof and Mechanisation

This chapter describes my mechanisation of WebAssembly's semantics in Isabelle/HOL [35], a proof of the type soundness property as stated in §2.7, and the issues that my mechanisation revealed in an earlier draft of the specification. This is the first full proof, mechanised or otherwise, of the soundness of the WebAssembly type system. A verified type checker and runtime interpreter built on top of the mechanisation are also described.

The current mechanisation is available under the name WasmCert-Isabelle [36] and is based on the W3C-published "WebAssembly 1.0" specification [26]. I have previously presented an earlier version of the mechanisation, proof, and verified executable artefacts [9]. This was based on a pre-print of the WebAssembly formal semantics, originally circulated in December 2016 [17]. This mechanisation found and corrected some errors in the draft semantics, which are also described below. These corrections were adopted into subsequent drafts.

3.1 Key lemmas of the type soundness proof

Many cases in the soundness proof fall out directly from unfolding and refolding of definitions. Below, I explain some of the representative and interesting cases by giving hand-proof analogues of the mechanised proofs, going through a few initial examples in particular detail. Each lemma is named according to its analogous mechanised lemma in WasmCert-Isabelle, proven as part of the overall type soundness proof.

3.1.1 Basic Lemmas

Lemma 3.1 (`e_type_consts`)

Assuming (1) $S; C \vdash v^* : tf$

we have $\exists t^* t_v^*.\ t_v^* = (\text{map typeof } v^*) \ \wedge \ (tf = t^* \to t^* t_v^*) \ \wedge \ S'; C' \vdash v^* : \epsilon \to t_v^*$

Proof. By induction on the definition of typing.

This is an inversion lemma which is used to recover facts about how a given typing judgement of the form $S; C \vdash v^* : tf$ must have been derived. In particular, because v^* is a list of values, it is typeable with $\epsilon \to (\text{map typeof } v^*)$ under any context, and tf must be some weakening of this type (representing an unaltered stack base). The lemma allows the type tf to be "pulled apart", so that we can refer to the separate components of the type. A version of this lemma is proven for each instruction in the WebAssembly language before proceeding with further proofs. Isabelle automatically derives inversion lemmas, but those are not sophisticated enough for our purposes, since we are effectively inverting two rules at once (the base typing rule, and possible applications of weakening).

Lemma 3.2 (`store_extension_refl`)

$S \prec_s S$

Proof. Follows straightforwardly from the definitions.

During the type soundness proof, there are some points where we need to prove that reduction respects store extension. In many cases, reduction does not alter the store, and this lemma allows such cases to be immediately discharged.

3.1.2 Preservation

We first establish two related auxiliary lemmas.

Lemma 3.3 (`inst_typing_store_extension_inv`)

Assuming (1) $S \vdash_i inst : C$

(2) $S \prec_s S'$

we have $S' \vdash_i inst : C$

Proof. Follows straightforwardly from the definitions. Intuitively, since S' is an extension of S, all elements of S pointed to by *inst* will have counterparts in S' with the same types.

Lemma 3.4 (`frame_typing_store_extension_inv`)

Assuming (1) $S \vdash_f F : C$

(2) $S \prec_s S'$

we have $S' \vdash_f F : C$

Proof. This follows from the definitions and `inst_typing_store_extension_inv`.

This lemma proves that frame typing is preserved by store extension. We later prove that reduction respects store extension, which implies (thanks to this lemma) that frame typing is preserved by reduction. This is a stepping stone towards the preservation property.

The bulk of the work in proving the preservation property for top-level configurations is carried out in first establishing a stronger lemma over program fragments. In this lemma, the configuration itself is not necessarily well-typed according to the configuration typing relation $\vdash_{\mathsf{c}}$, but is well-typed in some arbitrary enclosing label/frame context according to the instruction sequence typing relation $\vdash$.

Lemma 3.5 (`types_preserved_e2`)

Assuming (1) $S; F; e^* \hookrightarrow S'; F'; e'^*$

(2) $\vdash_{\mathsf{s}} S : \mathsf{ok}$

(3) $S \vdash_{\mathsf{f}} F : C$

(4) $S; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash e^* : tf$

we have $S \prec_{\mathsf{s}} S'$

$\vdash_{\mathsf{s}} S' : \mathsf{ok}$

$S' \vdash_{\mathsf{f}} F' : C$

$S'; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash e'^* : tf$

This lemma is proven by induction on the definition of reduction, taking C, tf, l_{arb}, and r_{arb} as arbitrary. The inductive hypothesis is only necessary to prove the cases corresponding to the congruence rules, since other rules are not inductively defined. We examine a few cases of the induction in detail. Note that in the mechanisation the induction is split between two sublemmas `reduce_store_extension` and `types_preserved_e1`, but for brevity each case is presented in a combined form here.

Recall the congruence rule for **frame**:

$$\frac{S; F; e^* \hookrightarrow S'; F'; e'^*}{S; \hat{F}; (\mathsf{frame}_n\ \{F\}\ e^*) \hookrightarrow S'; \hat{F}; (\mathsf{frame}_n\ \{F'\}\ e'^*)}$$

Here is the corresponding inductive case:

Lemma 3.6 (`types_preserved_e2` [**frame case**])

Assuming (1) $S; \hat{F}; (\mathbf{frame}_n\ \{F\}\ e^*) \hookrightarrow S'; \hat{F}; (\mathbf{frame}_n\ \{F'\}\ e'^*)$

(2) $\vdash_s S : \mathit{ok}$

(3) $S \vdash_f \hat{F} : C$

(4) $S; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash (\mathbf{frame}_n\ \{F\}\ e^*) : tf$

(5) $S; F; e^* \hookrightarrow S'; F'; e'^*$

(IH) $\forall C', tf', l'_{arb}, r'_{arb}.$
$(S; F; e^* \hookrightarrow S'; F'; e'^* \wedge$
$\vdash_s S : \mathit{ok} \wedge$
$S \vdash_f F : C' \wedge$
$S; C'[\text{label} := l'_{arb}, \text{return} := r'_{arb}] \vdash e^* : tf') \implies$
$(S \prec_s S' \wedge$
$\vdash_s S' : \mathit{ok} \wedge$
$S' \vdash_f F' : C \wedge$
$S'; C'[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash e'^* : tf')$

we have $S \prec_s S'$

$\vdash_s S' : \mathit{ok}$

$S' \vdash_f \hat{F} : C$

$S'; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash (\mathbf{frame}_n\ \{F'\}\ e'^*) : tf$

Proof.

taking existential witnesses t^*, t'^*, t^n we have

(6) $tf = (t^* \to t^*\ t'^*)$

(7) $S; (t^n) \vdash_{\mathsf{loc}} F; e^* : t'^*$

all by inverting (4) (note the inversion lemma must be proven manually by induction)

taking existential witness C' we have

(8) $S \vdash_{\mathsf{f}} F : C'$

(9) $S; C'[\text{return} := t^n] \vdash e^* : \epsilon \to t'^*$

all by inverting (7)

(10) $S \prec_{\mathsf{s}} S'$

(11) $\vdash_{\mathsf{s}} S' : \mathsf{ok}$

(12) $S' \vdash_{\mathsf{f}} F' : C'$

(13) $S'; C'[\text{return} := t^n] \vdash e'^* : \epsilon \to t'^*$

all by (IH) using (5), (2), (8), and (9)

(14) $S' \vdash_{\mathsf{f}} \hat{F} : C$

by `frame_typing_store_extension_inv` using (3) and (10)

(15) $S'; (t^n) \vdash_{\mathsf{loc}} F'; e'^* : t'^*$

by introduction using (12) and (13)

(16) $S'; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash (\textsf{frame}_n\ \{F'\}\ e'^*) : tf$
by introduction using (16) and weakening using (6)
QED by (10), (11), (14), and (16).

Here is the case corresponding to Wasm's reduction rule for **br**.

$$S; F; (\textsf{label}_n\ \{e^*\}\ L_k[v^n\ (\textsf{br}\ k)]) \hookrightarrow S; F; v^n\ e^*$$

Note that, because the rule has no premise, this case has no inductive hypothesis. An inner induction on the definition of L_k is required to determine the type of v^n.

Lemma 3.7 (`types_preserved_e2` [br case])
Assuming (1) $S; F; (\textsf{label}_n\ \{e^*\}\ L_k[v^n\ (\textsf{br}\ k)]) \hookrightarrow S; F; v^n\ e^*$
(2) $\vdash_s S : \textsf{ok}$
(3) $S \vdash_f F : C$
(4) $S; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash (\textsf{label}_n\ \{e^*\}\ L_k[v^n\ (\textsf{br}\ k)]) : tf$
we have $S \prec_s S$
$\vdash_s S : \textsf{ok}$
$S \vdash_f F : C$
$S; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash v^n\ e^* : tf$

Proof.
taking existential witnesses t^*, t'^*, t^n we have
(5) $tf = (t^* \to t^*\ t'^*)$
(6) $S; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash e^* : t^n \to t'^*$
(7) $S; C[\text{label} := (t^n)\ l_{arb}, \text{return} := r_{arb}] \vdash L_k[v^n\ (\textsf{br}\ k)] : \epsilon \to t'^*$
all by inverting (4) (again, inversion lemma must be proven by induction)

(8) $S; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash v^n : \epsilon \to t^n$
by rule induction on L_k using (7) and `e_type_consts`

(9) $S; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash v^n\ e^* : tf$
by introduction using (5), (6), and (8)
QED by `store_extension_refl`, (2), (3), and (9).

After proving each case of `types_preserved_e2` we have everything we need to prove the top-level preservation property.

Lemma 3.8 (`preservation`)

Assuming (1) $\vdash_{\mathsf{c}} S; F; e^* : t^*$

(2) $S; F; e^* \hookrightarrow S'; F'; e'^*$

we have $\vdash_{\mathsf{c}} S'; F'; e'^* : t^*$

Proof

(3) $\vdash_{\mathsf{s}} S : \mathsf{ok}$

(4) $S; \epsilon \vdash_{\mathsf{loc}} F; e^* : t^*$.

all by inverting (1)

(5) $S \vdash_{\mathsf{f}} F : C$

(6) $S; C[\text{return} := \epsilon] \vdash e^* : \epsilon \to t^*$

all by inverting (4)

(7) $\vdash_{\mathsf{s}} S' : \mathsf{ok}$

(8) $S' \vdash_{\mathsf{f}} F' : C$

(9) $S'; C[\text{return} := \epsilon] \vdash e'^* : \epsilon \to t^*$

as a special case of `types_preserved_e2`, using (2), (3), (5), and (6).

(10) $S'; \epsilon \vdash_{\mathsf{loc}} F'; e'^* : t^*$

by introduction with (8) and (9).

(11) $\vdash_{\mathsf{c}} S'; F'; e'^* : t^*$

by introduction with (7) and (10).

QED.

3.1.3 Progress

We can now turn our attention to the progress property. Just as with preservation, we must first prove a stronger lemma over program fragments, which are well-typed according to the instruction typing relation in some arbitrary label/frame context.

Lemma 3.9 (`progress_e`)

Assuming (1) $S; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash e^* : t^* \to t'^*$

(2) $C \vdash v^* : \epsilon \to t^*$

(3) $\forall L^k.\ e^* \neq L^k[\textsf{return}]$

(4) $\forall i\ L^k.\ e^* = L^k[\textsf{br}\ i] \implies i < k$

(5) $\forall v'^*.\ e^* \neq v'^*$

(6) $e^* \neq \textsf{trap}$

(7) $\vdash_s S : \textsf{ok}$

(8) $S \vdash_f F : C$

we have $\exists S'\ F'\ e'^*.\ S; F; v^*\ e^* \hookrightarrow S'; F'; e'^*$

and

assuming (9) $S; r^? \vdash v^*\ e^* : t'^*$

(10) $\forall L^k.\ e^* \neq L^k[\textsf{return}]$

(11) $\forall i\ L^k.\ e^* = L^k[\textsf{br}\ i] \implies i < k$

(12) $\forall v'^*.\ e^* \neq v'^*$

(13) $e^* \neq \textsf{trap}$

(14) $\vdash_s S : \textsf{ok}$

we have $\exists S'\ F'\ e'^*.\ S; F; v^*\ e^* \hookrightarrow S'; F'; e'^*$

The definition of (administrative) instruction typing is mutually recursive with local validity. Therefore we must prove this lemma using simultaneous *multi-predicate induction* over the definitions of instruction typing and local validity. Thankfully, Isabelle handles most of the book-keeping for this automatically. The top level typing judgement $\vdash_c S; F; e^* : t^*$ does imply that the configuration $S; F; e^*$ can execute one step, but this judgement is not general enough for us to successfully perform induction, so we cannot prove this directly. Instead we must perform induction on the more general typing judgement $S; C \vdash e^* : ft$. A key complication of the proof is that sequences of instructions which are well-typed with $S; C \vdash e^* : ft$ cannot necessarily execute by themselves, but *can* execute when embedded in a larger context represented by the type context C. I chose to deal with this by adding more assumptions to the induction to disallow these cases, and then handling them separately. There are three main reasons why a configuration $S; F; e^*$ may not reduce even if it is associated with a typing judgement $S; C \vdash e^* : t^* \to t'^*$.

First, the configuration may be terminal (i.e. a list of values, or a bare **trap**). Such configurations are well-typed, but are not expected to reduce, so these cases are discarded by premises (5), (6), (12), and (13). The requirement that only *non-terminal* configurations must progress is replicated in the top-level statement of the progress property.

Second, the input type of the judgement indicates the number of stack values that must be provided in order for the configuration to execute. So if $S; F; e^*$ is associated with the typing judgement $S; C \vdash e^* : t^n \to t'^*$, this is really saying that $S; F; v^n\ e^*$ can

execute, where $C' \vdash v^n : \epsilon \to t^n$. This is handled by adding the additional stack values to the conclusions of the induction, along with an assumption (2) about their well-typedness. In the top-level statement of the progress property, the input type of the configuration is required to be ϵ as part of the definition of $\vdash_{\mathsf{c}}$

Finally, the typing judgement $S; C[\text{return} := t^k] \vdash v^k$ **return** : tf may hold, but the code fragment v^k return on its own has no applicable reduction rule unless embedded a **frame** of arity k corresponding to return entry in the context C. This applies similarly to label and **br**. Such program fragments would form counter-examples to a naive inductive proof, so they are explicitly excluded to keep the induction regular (through premises (3), (4), (10), and (11)). These final cases are then handled by the following separate proofs showing that such cases cannot be well-typed according to the $\vdash_{\mathsf{loc}}$ typing definition (which types whole function bodies rather than program fragments under a context), and hence do not need to be considered when proving the top-level progress property:

Lemma 3.10 (`progress_e1`)
Assuming (1) $S; \epsilon \vdash_{\mathit{loc}} F; e^* : t^*$
we have $\forall L^k.\ e^* \neq L^k[\textbf{return}]$

Proof. By induction on the definition of the L context.

Lemma 3.11 (`progress_e2`)
Assuming (1) $S; \epsilon \vdash_{\mathit{loc}} F; e^* : t^*$
(2) $e^* = L^k[\textbf{br}\ i]$
we have $i < k$

Proof. By induction on the definition of the L context.

We can now use these lemmas together to prove the top-level progress property.

Lemma 3.12 (`progress`)
Assuming (1) $\vdash_{c} S; F; e^* : t^*$
we have $e^* = [\textbf{trap}] \vee (\exists v^*.\ e^* = v^*) \vee \exists S' F' e'.\ S; F; e^* \hookrightarrow S'; F'; e'^*$

Proof.
(2) $e^* \neq [\textbf{trap}]$
(3) $\nexists v^*.\ e^* = v^*$
assumption (discharge first two disjuncts of goal)

(3) $\vdash_{\mathsf{s}} S : \mathsf{ok}$
(4) $S; \epsilon \vdash_{\mathsf{loc}} F; e^* : t^*$.
all by inverting (1)

(5) $S \vdash_{\mathsf{f}} F : C$

(6) $S; C[\text{return} := \epsilon] \vdash e^* : \epsilon \to t^*$

all by inverting (4)

(7) $\exists S'\ F'\ e'^*.\ S; F; e^* \hookrightarrow S'; F'; e'^*$

as a special case of `progress_e`, `progress_e1`, `progress_e2`

QED.

3.1.4 Issues with the draft specification

When this proof was first attempted, it was against an early draft of the semantics and type system. Both in attempting to mechanise the draft specification, and in proving type soundness, a number of errors in the draft specification were discovered. Many of these errors were typos or other obvious definitional errors. Aside from these, three main issues were discovered which caused the original statement of type soundness to be false.

3.1.4.1 Trap propagation

Originally, the semantics omitted a rule to allow **trap** errors to propagate through straight-line code. This led to a trivial violation of the progress property, and was corrected by adding the rule

$$S; F; L_0[\mathsf{trap}] \hookrightarrow S; F; \mathsf{trap} \qquad \text{if } L_0[\mathsf{trap}] \neq \mathsf{trap}$$

3.1.4.2 Return typing

The **return** instruction was originally typed in a different way. The return element of the typing context C did not exist, and **return** was typed using the following rule:

$$\frac{\text{last}(C.\text{label}) = t^*}{C \vdash \mathsf{return} : t_1^*\ t^* \to t_2^*}$$

This was considered acceptable because entering a function introduces a **label** context (see the reduction rule for **call** in §2.4.5), so a "maximal **br**" will always terminate the current function. However, the reduction rule for **return** cannot locally determine how many enclosing label contexts it is situated inside, and is therefore specified as follows

$$S; \hat{F}; (\mathsf{frame}_n\ \{F\}\ L_k[v^n\ \mathsf{return}]) \hookrightarrow S; \hat{F}; v^n$$

Together, this meant that the judgement $\vdash_{\mathsf{c}} S; F; (\mathsf{label}_0\ \{\}\ \mathsf{return}) : \epsilon$ could be successfully typed, but the configuration could not reduce, creating a violation of the progress property. To fix this, it was necessary to add a distinguished return component to the typing context C, which is set when typing the body of a function or **frame** context. Note

that the the administrative instruction (**label**$_0$ {} **return**) cannot appear as the top-level reduct of a real program execution (since all real WebAssembly programs must begin with a function call, creating at least one **frame**) so the error was in spuriously allowing such a malformed intermediate reduct to be well-typed.

If this error had gone undiscovered, it would have become more of a practical issue if some future feature were to allow WebAssembly code with local control flow to be executed outside the context of a function call; for example if the initialising expressions of global variables were extended to allow **block**.

3.1.4.3 Host functions

The WebAssembly specification allows external host functions to be imported and called. The behaviour of these functions is outside the scope of WebAssembly's core semantics. However, they must still behave in a way which preserves the integrity of the WebAssembly state. For example, an invoked host function should not be permitted to dynamically change the type and value of a global variable from an i32 to an f64.

The draft WebAssembly semantics [17] originally did not formalise host function calls. A later draft of the specification made an attempt, but was missing the necessary axiomatic constraints as described in §2.4.5. In order to complete the type soundness proof, I formalised the correct constraints, and the official specification of host functions was later rewritten by Andreas Rossberg to incorporate these [26].

```
-- <instances>
record inst =
  types :: "tf list"
  funcs :: "i list"
  tabs :: "i list"
  mems :: "i list"
  globs :: "i list"
```

```
-- <function closures>
datatype cl =
  Func_native inst tf "t list" "b_e list"
| Func_host tf host
```

```
type_synonym tabinst = "(i option) list × nat option"
typedef meminst = "UNIV :: ((byte list) × nat option) set" ..
```

```
record global =
  g_mut :: mut
  g_val :: v
```

```
-- <store>
record s =
  funcs :: "cl list"
  tabs :: "tabinst list"
  mems :: "meminst list"
  globs :: "global list"
```

```
-- <frame>
record f =
  f_locs :: "v list"
  f_inst :: inst
```

Figure 3.1: Core definitions from the Isabelle model's runtime state.

```
-- <value types>
datatype t =
    T_i32 | T_i64
  | T_f32 | T_f64

-- <function types>
datatype tf =
  Tf "t list" "t list"
    ("_ '_> _" 60)

-- <immediate>
type_synonym i = nat

-- <packed types>
datatype tp =
    Tp_i8 | Tp_i16 | Tp_i32

-- <signedness flag>
datatype sx = S | U

-- <arithmetic ops>

datatype unop_i = ...
datatype unop_f = ...

datatype unop =
    Unop_i unop_i
  | Unop_f unop_f

datatype binop_i = ...
datatype binop_f = ...

datatype binop =
    Binop_i binop_i
  | Binop_f binop_f

datatype testop = Eqz

datatype relop_i = ...
datatype relop_f = ...

datatype relop =
    Relop_i relop_i
  | Relop_f relop_f

datatype cvtop =
    Convert | Reinterpret
```

```
-- <values>
datatype v =
    ConstInt32 i32
  | ConstInt64 i64
  | ConstFloat32 f32
  | ConstFloat64 f64

-- <basic instructions>
datatype b_e =
    Unreachable
  | Nop
  | Drop
  | Select
  | Block tf "b_e list"
  | Loop tf "b_e list"
  | If tf "b_e list" "b_e list"
  | Br i
  | Br_if i
  | Br_table "i list" i
  | Return
  | Call i
  | Call_indirect i
  | Get_local i
  | Set_local i
  | Tee_local i
  | Get_global i
  | Set_global i
  | Load t "(tp × sx) option" a off
  | Store t "tp option" a off
  | Current_memory
  | Grow_memory
  | EConst v ("C _" 60)
  | Unop t unop
  | Binop t binop
  | Testop t testop
  | Relop t relop
  | Cvtop t cvtop t "sx option"

-- <administrative instructions>
datatype e =
    Basic b_e ("$_" 60)
  | Trap
  | Invoke i
  | Label nat "e list" "e list"
  | Frame nat f "e list"
```

Figure 3.2: Core definitions from the Isabelle model's AST.

3.2 Details of the Mechanisation

The mechanisation of the WebAssembly semantics consists of ∼700 lines of non-comment, non-whitespace Isabelle/HOL, with the mechanisation of the type soundness proof consisting of ∼4900 further lines, and definitions and proofs related to the executable type checker and interpreter (see §3.3) consisting of ∼6600 further lines, not counting code handling the extraction to OCaml.

The core of the WebAssembly AST, typeset directly from the Isabelle mechanisation, is given in Fig. 3.2. The full enumerations of some numeric operations (such as `binop_i`) are elided for space. The mechanisation is broadly identical to the paper specification previously presented. However, as part of the process of mechanisation it was inevitable that some small changes to the definitions were needed, due to mismatches between the metatheory of Isabelle/HOL and the (implicit) metatheory of the paper specification; we discuss these below.

3.2.1 Reduction

The paper specification makes no distinction between administrative instructions and instructions, grouping both under the metavariable e. However, some expressions such as **loop** may not contain administrative instructions such as **invoke** in their body, and it is useful to make this distinction at the type level. The mechanisation declares regular instructions as the "basic instruction" datatype `b_e`, and then defines the instructions including the administrative instructions as the datatype `e`, with an explicit type constructor `Basic` :: `b_e` ⇒ `e` (abbreviated as `$`).

Recall that the paper specification puns values with **const** instructions. In the Isabelle mechanisation, values are defined with their own datatype `v`, while the **const** instruction is represented as `EConst v`, an instance of the `b_e` datatype. An abbreviation `C` is defined for the `EConst` type constructor, which is useful in concisely defining formal rules (see below).

The Isabelle mechanisation defines reduction using the inductive predicates

```
inductive reduce_simple :: e list ⇒ e list ⇒ bool
```

and

```
inductive reduce :: s ⇒ f ⇒ e list ⇒ s ⇒ f ⇒ e list ⇒ bool
```

abbreviated to

$$(\!|\texttt{es}|\!) \hookrightarrow (\!|\texttt{es'}|\!)$$

and

$$(\!|\texttt{s;f;es}|\!) \hookrightarrow (\!|\texttt{s';f';es'}|\!)$$

respectively. The types `s` and `f` refer to the store and the frame. They are defined as Isabelle records, with structures identical to those of the paper semantics (see Fig. 3.1). It is an Isabelle convention that types and variables should be in lowercase, in contrast to the WebAssembly paper specification.

The two levels of reduction in the mechanisation mirror a division in the paper specification, where the (unchanged) store and frame are elided in some reduction rules. In the paper specification, this is purely a syntactic shorthand. The mechanisation, however, must explicitly introduce the following rule relating `reduce_simple` and `reduce`.

$$\frac{e^* \hookrightarrow e'^*}{S; F; e^* \hookrightarrow S : F; e'^*}$$

This split approach is chosen both because it allows the "simple" rules to be more syntactically similar to those of the paper specification, and also because Isabelle/HOL can take a long time to process a single inductive predicate (such as `reduce`) with a large number of definitions.

As a concrete example, consider a paper reduction rule for **select**.

$$v_1\ v_2\ (\textbf{i32.const}\ l)(\textbf{select}) \hookrightarrow v_1 \qquad \text{where } l \neq 0$$

In the mechanisation, this rule is represented as a case of `reduce_simple`.

```
c ≠ 0 ⟹
  (|[$(C v1), $(C v2), $(C ConstInt32 c), $(Select)]|) ↪ (|[$(C v1)]|)
```

Note the explicit use of `C` to disambiguate values from other instructions, and the use of `$` to disambiguate "basic" instructions from administrative instructions.

As another example, consider the paper reduction rule for **global.set**.

$$S; F; v\ (\textbf{global.set}\ k) \hookrightarrow S'; F; \epsilon \qquad \begin{array}{l}\text{where } j = (F.\text{inst.iglobs})[k] \\ \qquad\qquad S' = S \text{ with } S'.\text{globs}[j] = v\end{array}$$

This is a good example of the specification's punning. Here, the metavariable v is used both as an *instruction* (in the redex) and as a *value* (when setting S'.globs$[j]$).

Here is the reduction rule for **global.set** as it appears in the Isabelle mechanisation.

```
definition supdate_glob ::  "s ⇒ inst ⇒ nat ⇒ v ⇒ s" where
  "supdate_glob s i k v =
     (let j = (inst.globs i)!k in
        s⦇globs := (globs s)[j:=((globs s)!j)⦇g_val := v⦈]⦈)"
...

-- <reduction rule for global.get>
"supdate_glob s (f_inst f) k v = s' ⟹
  ⦇s;f;[$(C v), $(Set_global k)]⦈ ↪ ⦇s';f;[]⦈"
```

The `supdate_glob` predicate in the premise replicates the side-condition of the original paper rule. Note that Isabelle's record update syntax is much more verbose than that of the paper semantics.

3.2.1.1 The label context

The largest divergence in the mechanised specification compared to the paper specification comes from Isabelle/HOL's inability to precisely type the paper specification's label evaluation contexts. Recall the definition of these contexts, together with a reduction rule for **br**.

$$\begin{aligned}(\text{label context}) \quad L_0[e^*] &::= v^*\ e^*\ e'^* \\ L_{k+1}[e^*] &::= v^*\ (\textbf{label}_n\ \{..\}\ L_k[e^*])\ e'^*\end{aligned}$$

$$S; F; (\textbf{label}_n\ \{e^*\}\ L_k[v^n\ (\textbf{br}\ k)]) \hookrightarrow S; F; v^n\ e^*$$

An L_k context contains precisely k nested labels. The value of k is used in a first-class manner when deciding whether a (**br** k) instruction can reduce. However, this constraint cannot be faithfully encoded in Isabelle/HOL's datatypes, because the system does not support dependent types.

Instead, the mechanisation defines a datatype for an arbitrary "context with hole", and uses a well-formedness predicate parameterised by k to express the nesting constraint.

```
datatype Lholed =
    -- <L0 case>
    LBase "v list" "e list"
    -- <L(k+1) case>
    | LRec "v list" nat "e list" Lholed "e list"


  inductive Lfilled ::  "nat ⇒ Lholed ⇒ e list ⇒ e list ⇒ bool" where
  -- <L0 case>
    L0:"lhole = (LBase vs es') ⟹
          Lfilled 0 lholed es (($C* vs) @ es @ es')"
  -- <L(k+1) case>
  | LN:"lhole = (LRec vs n es' l es'') ∧ Lfilled k l es lfillk ⟹
          Lfilled (k+1) lhole es (($C* vs) @ [Label n es' lfillk] @ es'')"
  ...

  -- <reduction rule for br>
  "length vs = n ∧ Lfilled k lholed (($C* vs) @ [$(Br k)]) LI ⟹
      (|[Label n es LI]|) ↪ (|($C* vs) @ es|)"
```

The inductive predicate `Lfilled` is defined to mirror the L_k context, representing the specification's type-level structure as an explicit predicate. Note that the Isabelle syntax $P \Longrightarrow Q$ defines an inductive rule with P as the premise and Q as the conclusion. The `Lfilled` definition needs two rules, each corresponding to a case of the L_k definition.

The `length vs = n ∧ Lfilled k lholed (($C* vs) @ [$(Br k)]) LI` precondition represents $L_k[v^n\ (\textbf{br}\ k)]$ in the paper reduction rule. The `lholed` parameter has the same structure as L_k, while the `k` parameter denotes the structure's depth (which is a type-level number in the implicit metatheory of the paper specification). In the paper-rule-based proofs described in §3.1, some steps involve induction on the definition of L_k. In the mechanisation, these same steps are accomplished by induction on the definition of `Lfilled`, with no additional complication. This is a standard approach in Isabelle/HOL when reasoning about structures which would otherwise naturally be dependently typed [37].

This reduction rule also highlights other small quirks of the mechanisation. Since `Label` is an administrative instruction, it is not lifted by `$`. List concatenation must be explicitly represented with the Isabelle concatenation operator `@` (we cannot replicate the paper specification's use of juxtaposition). The numbered superscript v^n in the paper rule, restricting the length of the value list, is represented as an explicit side-condition `length vs = n`. To lift the list of values `vs` to a list of instructions, the abbreviation `($C* vs)` is used, which simply maps `(λv. $C v)` over the list.

```
record limit_t =
  l_min :: nat
  l_max :: "nat option"
```

```
-- <table type>
type_synonym tab_t =
  "limit_t"
```

```
-- <memory type>
type_synonym mem_t =
  "limit_t"
```

```
-- <global type>
record tg =
  tg_mut :: mut
  tg_t :: t
```

```
-- <C context>
record t_context =
  types_t :: "tf list"
  func_t :: "tf list"
  global :: "tg list"
  table :: "tab_t list"
  memory :: "mem_t list"
  local :: "t list"
  label :: "(t list) list"
  return :: "(t list) option"
```

Figure 3.3: The Isabelle model's definition of the typing context C.

3.2.2 Type system

The basic instructions are typed using the inductive predicate:

```
inductive b_e_typing :: t_context ⇒ b_e list ⇒ tf ⇒ bool
```

abbreviated to

```
C ⊢ b_es :  tf
```

The definitions here match those given in §2.6 with the following exceptions.

WebAssembly's paper specification introduces some syntactic conventions which Isabelle/HOL has trouble representing. For example, consider the definition of binary operations:

$$
\begin{aligned}
binop_{\mathrm{i}N} &::= \ldots \\
binop_{\mathrm{f}N} &::= \ldots \\
\text{(instructions)}\quad e &::= \ldots \mid t.binop_t \mid \ldots
\end{aligned}
$$

The choice of which *binop* is allowed in the instruction is dependent on the value of the type annotation t. Isabelle/HOL does not support dependent types, and so we must unify the definitions under a single type as follows:

```
datatype binop_i = ...
datatype binop_f = ...

datatype binop =
    Binop_i binop_i
  | Binop_f binop_f

-- <basic instructions>
datatype b_e =
  ...
  | Binop t binop
  ...
```

When reasoning about well-typed WebAssembly programs, we include a well-formedness condition as part of the "well-typed" predicate, which enforces that an integer type-annotated *binop* may only contain integer operators. In concrete WebAssembly implementations, mismatching binary operators are prevented from occurring at the decoding stage. As a concrete example, here is the paper typing rule for **binop**.

$$\frac{}{C \vdash t.binop_t : t\ t \to t}$$

Here are the associated definitions in the mechanisation.

```
definition binop_t_agree :: "binop ⇒ t ⇒ bool" where
  "binop_t_agree binop t =
     (case binop of
        Binop_i _ ⇒ is_int_t t
      | Binop_f _ ⇒ is_float_t t)"
...

binop_t_agree op t ⟹
    C ⊢ [Binop t op] :  ([t,t] _> [t])
```

The metatheory of Isabelle/HOL also treats out-of-bounds list accesses differently, compared to that of the paper specification. For example, consider the following paper typing rule.

$$\frac{C.\text{global}[i] = t}{C \vdash \textbf{global.get}\ i : \epsilon \to t}$$

It is considered a type error if the static parameter of **global.get** is out-of-bounds of the list of declared global variables. The premise $C.\text{global}[i] = t$ enforces this implicitly; the typing derivation cannot be completed if the list access $C.\text{global}[i]$ is out-of-bounds.

In Isabelle/HOL, list access is a total function of type `'a list ⇒ nat ⇒ 'a`. If the provided index is out-of-bounds, an unspecified element of type `'a` is returned. This is possible because types in Isabelle/HOL are modelled as non-empty sets; every Isabelle/HOL type is *inhabited*.

However, this means that naively translating the typing rule into Isabelle/HOL would not be correct, because the rule would not bounds-check the static parameter of **global.get**. Instead, the bounds check must be explicitly added as a premise.

```
i < length (global C) ∧ (global C)!i = t ⟹
    C ⊢ get_global i : ([] _> [t])
```

As described in §2.7, the definition of typing must be extended to type the administrative instructions. The metatheory of the official specification is ambiguous as to precisely how this extension is defined. For the mechanisation, I explicitly introduce a separate inductive predicate, `e_typing`:

```
inductive e_typing :: s ⇒ t_context ⇒ b_e list ⇒ tf ⇒ bool
```

abbreviated to

```
S; C ⊢ es :  tf
```

combined with the following introduction rule (where `$*` abbreviates `map Basic`):

$$\frac{\texttt{C} \vdash \texttt{b\_es} : \texttt{tf}}{\texttt{S; C} \vdash \texttt{\$*\ b\_es} : \texttt{tf}}$$

3.2.3 Areas of the specification not mechanised

The WebAssembly formalism also includes a specification (not described within this thesis) of how the bytecode format is decoded into the WebAssembly module AST (Fig. 2.1). The mechanisation does not include this decoding step, instead beginning with the already decoded AST. The decoding step is well-separated from the rest of the specification: executing WebAssembly code cannot introspect its own representation.

In addition, the behaviours of WebAssembly's fundamental numeric operations (e.g. **add**, **div**) are not fully mechanised, with most operations (especially those dealing with IEEE 754 floating-point) left as uninterpreted functions. Since none of the type soundness proofs depend on these definitions, it would be trivial for them to be replaced with concrete formalisations of the relevant numeric operations as necessary.

3.3 Verified executable definitions

Isabelle/HOL allows inductive definitions to be extracted as executable OCaml code, if it is possible to automatically infer an executable definition [38]. However, the above definitions of the Wasm type system, and configuration reduction do not work well with this process, as some definitions are not obviously syntax-directed. For example, the typing rule for **br** (Fig. 2.6), due to its arbitrary output type. Instead, I define a separate type checker and interpreter for WebAssembly, in order to validate my mechanised model. The type checker is proven equivalent to the inductive typing relation. The interpreter is proven sound with respect to the definition of reduction.

The WebAssembly Community Group maintains an official reference implementation of WebAssembly in OCaml. My executable definitions were deliberately designed to be compatible with this implementation, so that portions of their implementation can be replaced with the verified extracted definitions. I provide a fork of the reference implementation which can freely switch between the official implementation, and my verified implementation [36].

3.3.1 Verified type checker

This section describes a type checker for WebAssembly implemented in Isabelle/HOL and proven equivalent to the previously described mechanisation of the language's type system. It is an explicit design goal of WebAssembly that code can be type checked in a single linear pass. However, the type system of §2.6 is not trivially executable in this manner. This is because of the type system's approach to typing dead code which follows sequentially after a **br**, **return**, **br_table**, or **unreachable** instruction. The proof of correctness of the type checker, once defined, is uncomplicated, but the definition itself is subject to several complications because of the above.

Consider the typing rule for **return**.

$$\frac{C.\text{return} = t^*}{C \vdash \textbf{return} : t_1^*\ t^* \to t_2^*}$$

The result type t_2^* is picked arbitrarily. However, the type system effectively assumes that the choice is made angelically [39], since it is still possible for subsequent, syntactically dead code to cause a type error, depending on the type picked. Consider the type derivation for the following code fragment, assuming $C.\text{return} = \textsf{i64}$

$$\dfrac{\dfrac{}{C \vdash (\textbf{i64.const}\ 5) : \epsilon \to \textsf{i64}} \quad \dfrac{C.\text{return} = \textsf{i64} \quad {}^*\textit{angelically pick result type here}}{C \vdash (\textbf{return}) : \textsf{i64} \to \textsf{f32}\ \textsf{f32}}}{\dfrac{C \vdash (\textbf{i64.const}\ 5)\ (\textbf{return}) : \epsilon \to \textsf{f32}\ \textsf{f32} \qquad \dfrac{}{C \vdash \textbf{f32.add} : \textsf{f32}\ \textsf{f32} \to \textsf{f32}}}{C \vdash (\textbf{i64.const}\ 5)\ (\textbf{return})\ (\textbf{f32.add}) : \epsilon \to \textsf{f32}}}$$

Imagine writing an algorithm that, given the context C, linearly walks this list of instructions, determining whether it is well-typed by typing a progressively longer prefix of the list. When typing the prefix (**i64.const** 5) (**return**), the algorithm must choose the right output type to ensure that the suffix of the list is typeable. In this case it must pick f32 f32. However it does not have the knowledge necessary to do this without looking ahead in a potentially unbounded way. Consider that if the suffix was (**f32.add**)(**f32.add**), then the output type of the prefix would need to be f32 f32 f32, and so on.

Now consider the following code fragment.

(**i64.const** 5) (**return**) (**f32.add**) (**i64.add**)

This fragment is ill-typed, because the prefix (**i64.const** 5) (**return**) (**f32.add**) must have a output type with f32 at the head, which conflicts with the input type required by (**i64.add**).

A related complication comes from the **select** instruction. Consider the following typing derivation.

$$\dfrac{\dfrac{}{C \vdash (\textbf{i64.const}\ 5) : \epsilon \to \mathsf{i64}} \quad \dfrac{C.\mathrm{return} = \mathsf{i64} \quad {}^*\textit{angelically pick result type here}}{C \vdash (\textbf{return}) : \mathsf{i64} \to t\ t\ \mathsf{i32}}}{C \vdash (\textbf{i64.const}\ 5)\ (\textbf{return}) : \epsilon \to t\ t\ \mathsf{i32}} \quad \dfrac{}{C \vdash \textbf{select} : t\ t\ \mathsf{i32} \to t}}{C \vdash (\textbf{i64.const}\ 5)\ (\textbf{return})\ (\textbf{select}) : \epsilon \to t}$$

Here, the ultimate output type is arbitrary but guaranteed to be non-empty. The **select** instruction is the only (unfortunate) example in WebAssembly where the output type of the instruction is not precisely determined. Consider the following code.

```
block (ϵ → ϵ)
    (br 0) select
end
```

This code is ill-typed, because the body of the block has a non-empty type.

Writing a linear type checking algorithm which types prefixes of the instruction list purely with WebAssembly types is not possible. Instead, the algorithm must type the stack using an extended syntax which incorporates polymorphic symbols, essentially a simple constraint system which defers concretisation of the output type.

The extended types used by the type checker are as follows.

```
datatype ct =
    TAny
  | TSome t

datatype checker_type =
    TopType "ct list"
  | Type "t list"
  | Bot
```

The type checker checks a list of instructions by iterating over them, keeping track of the state of the stack at each program point as a `checker_type`. The regular case is `Type "t list"`, which denotes a regular list of Wasm types (t^*), with no polymorphism. The case `TopType "ct list"` represents a stack type with an unconstrained base, and a head of potentially polymorphic types. After processing an unconditional control flow operation such as **return**, which the Wasm type system gives an arbitrary result type, the current `checker_type` is instead set to the polymorphic type `TopType []`. This represents a stack with entirely unconstrained size and contents. Subsequent operations may append precise types in the form `TSome t`. As shown above, the **select** instruction, when applied to a polymorphic stack, will produce a stack which is known to be non-empty, but the type of the top element is not precisely determined. This is represented using the `TAny` type. Finally, the `Bot` type represents an ill-typed stack.

The type checker mirrors the form of the typing judgement $C \vdash e^* : t^n \to t^m$. At the top level, the function `b_e_type_checker` is given a typing context `C` representing the current scope information, a list of instructions to type (`es`), and an expected function type `tn _> tm`. Starting with the input type `tn` as the current state of the stack, each instruction is processed in turn by the recursive function `check`. Once all instructions are processed, the result type (which may be a polymorphic type of the extended syntax above) is checked for compatibility with the desired return type. An essentially identical strategy (and extended type syntax) is used by the official reference implementation.

Before the code for the type checker is shown, it is necessary to discuss some auxiliary definitions.

```
fun ct_compat :: "ct ⇒ ct ⇒ bool" where
  "ct_eq (TSome t) (TSome t') = (t = t')"
| "ct_eq TAny _ = True"
| "ct_eq _ TAny = True"

definition ct_list_compat :: "ct list ⇒ ct list ⇒ bool" where
  "ct_list_compat ct1s ct2s = list_all2 ct_compat ct1s ct2s"

definition ct_suffix ::  "ct list ⇒ ct list ⇒ bool" where
  "ct_suffix xs ys = (∃ as bs.  ys = as@bs ∧ ct_list_compat bs xs)"
```

This defines a way of comparing `ct` values in terms of the underlying stack types they represent. The `ct_compat` predicate returns true if the two types are compatible. That is, if they are equal, or if one is more general than the other (`TAny`).

This is generalised to compatibility of `ct_list` by `ct_list_compat`, and then to compatibility of suffixes by `ct_suffix`, which is true iff its first argument is compatible with a suffix of its second argument.

```
fun consume :: "checker_type ⇒ ct list ⇒ checker_type" where
  "consume (Type ts) cons = (if ct_suffix cons (to_ct_list ts)
                               then Type (take (length ts - length cons) ts)
                               else Bot)"
| "consume (TopType cts) cons = (if ct_suffix cons cts
                                   then TopType (take (length cts - length cons) cts)
                                   else (if ct_suffix cts cons
                                           then TopType []
                                           else Bot))"
| "consume _ _ = Bot"

fun produce :: "checker_type ⇒ checker_type ⇒ checker_type" where
  "produce (TopType ts) (Type ts') = TopType (ts@(to_ct_list ts'))"
| "produce (Type ts) (Type ts') = Type (ts@ts')"
| "produce (Type ts') (TopType ts) = TopType ts"
| "produce (TopType ts') (TopType ts) = TopType ts"
| "produce _ _ = Bot"

fun type_update :: "checker_type ⇒ ct list ⇒ checker_type ⇒ checker_type" where
  "type_update curr_type cons prods = produce (consume curr_type cons) prods"
```

These definitions form the core of the type checking procedure. Most instructions simply produce and consume values to/from the stack type according to their typing rule. Thus, typing the instruction amounts to checking the premises of its typing rule, then calling `type_update` with the appropriate arguments, which uses `consume` and `produce` to update the stack. The `consume` function behaves differently depending on whether the current stack type is a precise `Type`, or a `TopType` (representing the unconstrained base caused by typing dead code). A precise `Type` must provide all the types being consumed. However, a `TopType` only needs to provide a suffix of the types being consumed; its unconstrained base indicates that dead code can be validated as though the type stack base contains the required concrete types, mirroring the angelic choice previously described. If the required types cannot be consumed, `Bot` is returned. Otherwise, the updated stack type is returned.

The `produce` function appends types to the head of the stack type. However, it must also handle the case of an instruction such as **unreachable** producing a `TopType` (representing its arbitrary output type), in which case the current stack type is entirely replaced. We can now consider the top-level type checking functions.

```
fun b_e_type_checker :: "t_context ⇒  b_e list ⇒ tf ⇒ bool"
and check :: "t_context => b_e list ⇒ checker_type ⇒ checker_type"
and check_single :: "t_context ⇒  b_e ⇒ checker_type ⇒ checker_type" where
  "b_e_type_checker C es (tn _> tm) = c_types_agree (check C es (Type tn)) tm"
| "check C es ts =
      (case es of
        [] => ts
      | (e#es) => (case ts of
                          Bot => Bot
                        | _ => check C es (check_single C e ts)))"

  -- <num ops>
| "check_single C_ (C v) ts = type_update ts [] (Type [typeof v])"
| "check_single C (Unop t op) ts =
      (if unop_t_agree op t
        then type_update ts [TSome t] (Type [t])
        else Bot)"
  -- <block>
| "check_single C (Block (tn _> tm) es) ts =
      (if (b_e_type_checker (C⦇label := ([tm] @ (label C))⦈) es (tn _> tm))
        then type_update ts (to_ct_list tn) (Type tm)
        else Bot)"
  -- <other cases ...>
```

The `check_single` function contains the main logic for typing individual instructions. Given a typing context, an instruction to be typed, and a current stack type (in the extended syntax), the function returns the output stack type. If the instruction would make the stack ill-typed, the stack type `Bot` is returned. This can happen either because a pre-condition of the typing rule is violated, or because the input stack type is of the wrong shape. The `type_update` function handles this latter case. Given a stack type, and a number of types to consume and produce, it will either return the updated stack, or `Bot` if the provided stack does not have the right types to consume. If the provided stack type is a `TopType`, consuming values from the unconstrained base will always succeed.

The `check` function contains the iterative logic. It acts on a list of instructions. If the current stack type is `Bot`, type checking returns immediately with that result. Otherwise, it processes each instruction in turn with `check_single` until a final result is reached.

The `b_e_type_checker` function contains the top level type checking procedure. Given a typing context, a list of instructions, and a type annotation to be checked, the `check` function is called, using the input type of the type annotation as the initial stack type. Once this call has produced an output stack type in the extended syntax, the `c_types_agree` function checks to see whether this stack type is compatible with the output type of the provided type annotation (`tm`). When the `check_single` function is typing a nested instruction such as **block**, it will recursively call `b_e_type_checker` in order to check the instruction's body.

Here are some examples of how the type checker operates:

	`Type []`
(**i32.const** 1)	`Type [T_i32]`
(**i32.const** 2)	`Type [T_i32, T_i32]`
(**i32.add**)	`Type [T_i32]`

In this example, the code fragment has been successfully typed with $\epsilon \to$ i32.

	`Type []`
return	`TopType []` (if `return C = []`)
(**i32.const** 2)	`TopType [T_i32]`
(**i32.add**)	`TopType [T_i32]`

In this example, the result type has a polymorphic base, and is guaranteed to have i32 at its head. The code fragment has been typed $\epsilon \to t^*$ i32, where t^* can be arbitrarily picked.

As hinted earlier, the **select** case must be handled specially. Unfortunately, it cannot be expressed in terms of `type_update`. Here is how the **select** case of `check_single` must be defined.

```
fun select_return_top :: "[ct list] => ct => ct => checker_type" where
  "select_return_top ts ct1 TAny = TopType ((take (length ts - 3) ts) @ [ct1])"
| "select_return_top ts TAny ct2 = TopType ((take (length ts - 3) ts) @ [ct2])"
| "select_return_top ts (TSome t1) (TSome t2) =
      (if (t1 = t2)
        then (TopType ((take (length ts - 3) ts) @ [TSome t1]))
        else Bot)"

fun type_update_select :: "checker_type => checker_type" where
  "type_update_select (Type ts) =
      (if (length ts >= 3 ∧ (ts!(length ts-2)) = (ts!(length ts-3)))
        then consume (Type ts) [TAny, TSome T_i32]
        else Bot)"
| "type_update_select (TopType ts) =
      (case length ts of
        0 => TopType [TAny]
      | Suc 0 => type_update (TopType ts) [TSome T_i32] (TopType [TAny])
      | Suc (Suc 0) => consume (TopType ts) [TSome T_i32]
      | _  => type_update (TopType ts) [TAny, TAny, TSome T_i32]
                    (select_return_top ts (ts!(length ts-2)) (ts!(length ts-3))))"
| "type_update_select _ = Bot"

  -- <... cases of check_single>
  -- <select>
| "check_single C (Select) ts = type_update_select ts"
```

At a high level, the typing of **select** proceeds by peeking backwards in the type stack. Recall that the typing rule for **select** is:

$$\overline{C \vdash \textsf{select} : t\ t\ \mathsf{i32} \to t}$$

In the case that the current stack type is `Type ts`, the algorithm must ensure that the suffix of `ts` contains two types which are the same, and a `T_i32`, and that the output type matches the input types. In addition, if the current stack type is a `TopType`, we must carefully ensure that the resulting type is correct if unconstrained types are consumed. It is convenient to split this case into subcases, depending on the length of the head of the `TopType`, in order to simplify the proof of correctness.

- If the stack type is of the form `TopType []`, then the result is `TopType [TAny]`.
- If the stack type is of the form `TopType [ct]`, then if `ct` is compatible with `T_i32`, then the result is `TopType [TAny]`, otherwise the result is `Bot`.
- If the stack type is of the form `TopType [ct1, ct2]`, then if `ct2` is compatible with `T_i32`, then the result is `TopType [ct1]`, otherwise the result is `Bot`.

- The final case is defined in a helper function `select_return_top`, and is the most complicated. If the stack type is of the form `TopType cts@[ct1, ct2, ct3]`, then if `ct1` and `ct2` are compatible with each other, and if `ct3` is compatible with `T_i32`, then the result is `TopType cts@[ct]`, where `ct` is the most precise type out of `ct1` and `ct2` (i.e, if `ct1` is `TAny`. then `ct = ct2`). Otherwise, the result is `Bot`.

Having carefully set up all the definitions, the proof of correctness of the type checker is straightforwardly accomplished by exhaustively case-splitting everything, with a simple induction to handle recursive cases like **block**. The following equivalence is proven in Isabelle/HOL.

Lemma 3.13 (Type Checker Correctness)

```
"(C ⊢ es :  tf) = (b_e_type_checker C es tf)"
```

It is notable that the vast majority of engineering and proof effort in defining and verifying this type checker comes from cases which arise purely when typing dead code. The choice to allow dead code in WebAssembly was made to support naive streaming producers which may not wish to track properties such as syntactic deadness. If dead code was not validated, then the type checker would be far simpler, as the entire constraint system approach could be discarded, and the intermediate stack types could be faithfully stated using WebAssembly's core type syntax. The implementation strategy needed to validate dead code as currently required by the spec has been highlighted by some tool maintainers as a source of bugs [40]. I have presented a potential relaxing of the WebAssembly type system to the Community Group based on the following typing rules:

$$\frac{C.\text{label}[i] = t^*}{C \vdash \textbf{br}\ i : t_1^*\ t^* \to \bot} \qquad \frac{(C.\text{label}[i] = t^*)^+}{C \vdash \textbf{br\_table}\ i^+ : t_1^*\ t^*\ \text{i32} \to \bot} \qquad \frac{C.\text{return} = t^*}{C \vdash \textbf{return} : t_1^*\ t^* \to \bot}$$

$$\frac{}{C \vdash \textbf{unreachable} : t^* \to \bot} \qquad \frac{}{C \vdash e^* : \bot \to t^*}$$

The addition of an explicit $\bot$ stack type would effectively mean that validation of dead code is skipped, removing the need for the more complicated type stack when type checking. This proposal has currently reached standardisation phase 2 of 4 [41], and further discussions are ongoing.

3.3.1.1 Module validation

The executable type checker defined above is used as the basis for an executable module validator, implementing the judgement defined in §2.8.1. As mentioned, the original judgement defines a circular definition of C, and the executable definition breaks the circularity by collecting all the function type annotations as a separate pre-pass. We prove (simply) that this executable definition is equivalent to the judgement of §2.8.1.

3.3.2 Verified interpreter

I also define an executable interpreter, which is proven sound with respect to the mechanised definition of reduction. Here, the main complication is the Wasm specification's definition of control flow operations such as **br**. Consider these reduction rules involving **label**.

$$\begin{aligned}\text{(label context)} \quad L_0[e^*] &::= v^*\ e^*\ e'^* \\ L_{k+1}[e^*] &::= v^*\ (\textbf{label}_n\ \{..\}\ L_k[e^*])\ e'^*\end{aligned}$$

$$\frac{S; F; e^* \hookrightarrow S'; F'; e'^*}{S; F; L_k[e^*] \hookrightarrow S'; F'; L^k[e'^*]}$$

$$S; F; (\textbf{label}_n\ \{e^*\}\ v^*) \hookrightarrow S; F; v^*$$

These rules naturally lend themselves to a straightforward, recursive definition of execution. When a **label** construct is executed one step, this is defined in terms of executing its body (which may itself contain a **label**) one step, unless its body has already reduced to a list of values, in which case the values are propagated out of the **label**. However, consider the rule for the reduction of **br**.

$$S; F; (\textbf{label}_n\ \{e^*\}\ L_k[v^n\ (\textbf{br}\ k)]) \hookrightarrow S; F; v^n\ e^*$$

This breaks the regular recursive structure. By blindly recursing into the body, the outer **label** which determines where the inner **br** will jump to is discarded. In order to allow a regular recursive definition of execution, it is necessary to introduce intermediate computation results which represent a **br** instruction in the process of breaking out of its enclosing labels.

Reduction in the mechanisation is defined as

```
inductive reduce :: s ⇒ f ⇒ e list ⇒ s ⇒ f ⇒ e list ⇒ bool
```

Recall that the `e list` components represent both the current value stack, and the instructions to be executed. In the interpreter, a single runtime state is represented as follows:

```
type_synonym config_tuple = "s × f × v list × e list"
```

In this definition, the value stack is explicitly kept separate to reduce unnecessary list processing during execution. A single step of execution in the interpreter is defined as returning a `res_tuple`.

```
type_synonym res_tuple = "s × f × res_step"
```

```
datatype res_step =
  RSCrash
| RSBreak nat "v list"
| RSReturn "v list"
| RSNormal "v list" "e list"
```

The result of a single step of execution may correspond either to a regular reduction step (denoted by `RNormal`, the value stack is again kept separate), or alternatively the result may indicate that this execution step contains a **br** or **return** instruction which is targetting an outer context (`RSBreak` and `RSReturn` respectively), or finally, the result may indicate an unexpected error through `RSCrash` (which should never occur in a well-typed program).

```
abbreviation vs_to_es :: " v list ⇒ e list" where
  "vs_to_es v ≡ $C* (rev v)"

function (sequential)
  run_step :: "depth ⇒ config_tuple ⇒ res_tuple" where
  "run_step d (s,f,(ves, es)) =
      (case es of
        [] ⇒ (s,f, crash_error)
      | e#es' ⇒ if e_is_trap e then
                  if (es' != [] ∨ ves != []) then
                    (s, f, RSNormal [] [Trap])
                  else
                    (s, f, crash_error)
                else
                  case e of
                    -- <unops>
                    $(Unop t op) ⇒
                      (case ves of
                        v#ves' ⇒ (s, f, RSNormal ((app_unop op v)#ves') es')
                       | _ ⇒ (s, f, crash_error))
                   -- <more cases...>
```

Figure 3.4: Top-level definition for a single step of the interpreter

The `run_step` function of Fig. 3.4 performs a single step of execution. The `depth` parameter places a limit on the number of nested function calls that are permitted, as the official Wasm test suite includes a test that requires concrete implementations to terminate with an error if the call depth goes over an implementation-defined limit. Recall that in the formal definition of reduction, the value stack is left implicit as the list of leading v instructions. In the interpreter, the value stack is explicitly split off as `ves` so that it can be handled more efficiently. The following relationship holds (we will discuss its proof later). Note that the value stack is reversed, compared to its treatment in the relational reduction rules; `vs_to_es ves` is defined as `$C* (rev ves)`.

```
run_step d (s,f,(ves,es)) = (s',f',RSNormal ves' es') ⟹
        (|s;f;(vs_to_es ves)@es|) ↪ (|s';f';(vs_to_es ves')@es'|)
```

Considering the cases of `run_step`, there should always be at least one non-value instruction in the reduct (otherwise no reduction should be performed), so if the list of remaining non-value instructions `es` is empty, the result is a crash error. If the instruction to be executed is a **trap**, the value stack and all subsequent instructions are discarded; the overall result of the reduction is a bare **trap**. To maintain equivalence with the reduction rules, an already bare **trap** with no value stack or instructions to discard crashes rather than reducing to itself. If there is at least one non-trap instruction to execute, then a case split occurs to identify the instruction and carry out the relevant reduction. In most cases, this corresponds precisely to the behaviour of the relational reduction rule. For example, the listing above depicts the execution of the **unop** instruction. the result `(s, f, RSNormal ((app_unop op v)#ves') es')` depicts the value stack updated with the result of the **unop**, with `es'` representing the remaining instructions to be executed.

The interpreter must be more complicated in cases involving the results `RSBreak` or `RSReturn`, which indicate that the current invocation of `run_step` is part of a recursion, and the result must be dealt with by an outer call. Here are some relevant cases of `run_step` illustrating this.

```
-- <br>
| $Br j ⇒
    (s, f, RSBreak j ves)
-- <...>
-- <label>
| Label ln les es ⇒
    if es_is_trap es
      then
        (s, f, res_trap ves es')
       else
         (case (split_vals_e es) of
            (lsves, []) ⇒ (s, f, RSNormal ((rev lsves)@ves) es')
          | (lsves, lses) ⇒
              let (s', f', res) = run_step d (s, f, (rev lsves, lses)) in
              (case res of
                 RSBreak 0 bvs ⇒
                   if (length bvs >= ln)
                     then (s', f', RSNormal ((take ln bvs)@ves) (les@es'))
                     else (s', f', crash_error)
               | RSBreak (Suc n) bvs ⇒
                   (s', f', RSBreak n bvs)
               | RSReturn rvs ⇒
                   (s', f', RSReturn rvs)
               | RSNormal lsves' lses' ⇒
                   (s', f',
                   RSNormal ves ((Label ln les ((vs_to_es lsves')@lses'))#es'))
               | RSCrash c ⇒ (s', f', RSCrash c)))
```

The **br** case simply returns an `RSBreak` result which records the size of the break, and the current stack values. An `RSBreak j ves` result represents a **br** instruction that is breaking out of the `j`-th enclosing label. The `ves` component will be needed to satisfy the arity of that label (see below).

The **label** case must handle multiple reduction rules. If the body of the **label** is a trap or pure list of values, the **label** is discarded and the body is returned. In the case that the body of the **label** must be executed a step, a recursive call to `run_step` is performed. If the result of that call is a regular `RSNormal`, the result of the **label** reduction is simply the same **label** with the reduced body. However, if the result is an `RSBreak`, it must be handled specially. A `RSBreak (Suc n) bvs` result indicates that an outer label is being targetted. Therefore the result of the current call is `RSBreak n bvs`, reflecting the fact that one more label has been passed through. If the result is `RSBreak 0 bvs`, then this

result came from an inner **br** that targetted the current label. If the length of `bvs` is large enough to satisfy the arity of the label (guaranteed for well-typed programs), then that sublist of `bvs` is appended to head of the value stack, and the label is discarded, replaced by its continuation component `les`.

The case for **frame** handles the `RSReturn` result in the same way that the **label** case must handle an `RSBreak 0` result.

3.3.2.1 Proving the interpreter sound

Cases such as **unop** where the interpreter simply performs the appropriate calculation and returns an `RSNormal` can be trivially discharged, since they correspond precisely to the relevant reduction rule. However, some auxilliary lemmas must be proven in order to relate `RSBreak` result to reduction of the enclosing **label**.

As a first step, it is necessary to define a modified version of the `Lfilled` context in order to state some auxilliary results.

```
inductive Lfilled_exact ::  "nat ⇒ Lholed ⇒' e list ⇒ e list ⇒ bool" where
-- <L0 case>
  L0:"lhole = (LBase [] []) ⟹
        Lfilled_exact 0 lholed es es"
-- <L(k+1) case>
| LN:"lhole = (LRec vs n es' l es") ∧ Lfilled_exact k l es lfillk ⟹
        Lfilled_exact (k+1) lhole es (($C* vs) @ [Label n es' lfillk] @ es")"
```

This is an Isabelle definition of an inductive predicate `Lfilled_exact`, with an inductive case identical to that of `Lfilled`, but a different base case, which exposes the entire contents of the innermost label. Contrast this with the definition of `Lfilled` previously shown (§3.2.1.1) where the hole may occur as part of a larger sequence of instructions within the innermost label.

Lemma 3.14 (`run_step_return_imp_lfilled`)

Assuming (1) `"run_step d (s,f,ves,es) = (s', f', RSReturn res)"`

we have

```
"s = s' ∧
 f = f' ∧
(∃ n lfilled es_c.
    Lfilled_exact
        n
        lfilled
        ((vs_to_es res) @ [$Return] @ es_c)
        ((vs_to_es ves)@es))"
```

Proof. By induction on the definition of `run_step`.

This lemma shows that, if an invocation of `run_step` results in `RSReturn res`, then the executed code `es` must be made up of nested **label** constructs such that the body of the innermost **label** is precisely `((vs_to_es res) @ [$Return] @ es_c)`, for some existentially quantified `es_c`. Note that this lemma is stronger than a hypothetical version of the same lemma with `Lfilled` in the conclusion, which could only state that `((vs_to_es res) @ [$Return] @ es_c)` is *part of* the innermost label body.

A slightly more complicated version of this lemma exists for `RSBreak`.

Lemma 3.15 (`run_step_break_imp_lfilled`)

Assuming (1) `"run_step d (s,f,ves,es) = (s', f', RSBreak n res)"`

we have

```
"s = s' ∧
 f = f' ∧
(∃ n' lfilled es_c.
    n' ⩽ n ∧
    Lfilled_exact
        (n' - n)
        lfilled
        ((vs_to_es res) @ [$br n'] @ es_c)
        ((vs_to_es ves)@es))"
```

Proof. By induction on the definition of `run_step`.

If an invocation of `run_step` results in `RSBreak n res`, then the executed code `es` must be made up of (`n' - n`) nested **label** constructs such that the body of the innermost **label** is precisely `((vs_to_es res) @ [$break n'] @ es_c)`, for some existentially quantified `n'` and `es_c`. This lemma is analogous to the previous one for `RSReturn`, while additionally relating the depth of the current context of nested labels to the index of the **br** instruction which must have caused the `RSBreak` result.

Lemma 3.16 (`run_step_sound`)

Assuming (1) `"run_step d (s,f,(ves,es)) = (s',f',RSNormal ves' es')"`

we have `⦇s;f;(vs_to_es ves)@es⦈ ↪ ⦇s';f';(vs_to_es ves')@es'⦈`

As discussed above, the vast majority of cases can be discharged trivially. Cases where the result of a recursive `run_step` call is not `RSNormal` are proven by applying the lemmas detailed above, which show that the program fragment being executed must be embedded within a larger context for which reduction according to the relational rules is possible.

3.4 Experimental validation and testing

The official reference implementation of WebAssembly, written in OCaml, comes with an extensive test suite, written in a bespoke scripting language that is an extension of the WebAssembly text format. This section describes the work necessary to take the previously described verified type checker and interpreter, and build an end-to-end WebAssembly implementation capable of running these tests. Using Isabelle's extraction feature [42], we obtain OCaml versions of the Isabelle definitions of the type checker and interpreter. The official reference implementation is then used to fill in missing corners as detailed below, to allow end-to-end execution.

3.4.1 Numerics

As mentioned earlier, the WebAssembly mechanisation does not attempt to specify the semantic details of WebAssembly's numeric operations (e.g. **add**, **div**), since none of the previously described correctness proofs depend on them. Instead numeric operations are specified as uninterpreted functions, which can be replaced with concrete implementations when extraction to OCaml is performed. It would be trivial to swap in any alternative Isabelle-level definition of numerics, but this is left for future work. Instead, extraction is configured to use the official OCaml reference interpreter's numeric definitions.

3.4.2 Decoding

We do not specify the decoding of WebAssembly programs in Isabelle – all definitions manipulate an already-decoded AST. Beyond this, to run the official test suite, it would not be enough to reproduce the WebAssembly specification's syntax rules in Isabelle, we would also need to implement the testing-specific extensions used by the reference implementation. Instead, we use the existing reference implementation's decoder.

3.4.3 Testing

Once all of this has been done, we are able to run the official WebAssembly test suite, exercising the extracted verified definitions. Running these tests initially revealed a handful of typo-level bugs in the "shim code" which connected the Isabelle definitions to the decoding code of the OCaml reference implementation. Once these had been corrected, a number of tests which involved heavy manipulation of the WebAssembly heap were found to terminate with an OCaml stack overflow. On closer inspection, this was due to some Isabelle list-manipulation functions being defined in a way that was not tail-recursive. For example, the Isabelle function for producing a list containing n copies of a given element is as follows:

```
primrec replicate :: "nat ⇒ 'a ⇒ 'a list" where
replicate_0: "replicate 0 x = []" |
replicate_Suc: "replicate (Suc n) x = x # replicate n x"
```

To create a new section of zero-initialised WebAssembly memory (represented as a `byte list`), the Isabelle mechanisation calls `replicate (n*(2^16)) (0::byte)`, where `n` is the desired number of pages. Since each element of the list requires a further-nested call to `replicate`, the extracted code quickly overflows the OCaml call stack.

This problem can be solved by telling Isabelle to use an alternative implementation of `replicate` when extracting to OCaml. Unlike the previously described unverified substitution of numerics for their OCaml implementations, this can be done in a fully verified way, by defining the alternative tail-recursive function `replicate_tr` in Isabelle, proving it equivalent to `replicate`, and commanding Isabelle to replace instances of `replicate` with `replicate_tr` during extraction.

```
primrec replicate_tr :: "nat ⇒ 'a ⇒ 'a list ⇒ 'a list" where
"replicate_tr 0 x acc = acc" |
"replicate_tr (Suc n) x acc = replicate_tr n x (x#acc)"
```

Once this final issue had been solved, the extracted Isabelle code (combined with the OCaml reference implementation's decoder and numeric definitions) passes all tests. Although memory manipulation tests no longer overflow the stack, our list-based representation of memory is still very inefficient: each memory access essentially traverses the entire list in linear time, and takes a copy of the list if some part of the memory must be mutated. Some tests which run instantaneously on the reference implementation take several minutes to run when switching to the Isabelle-extracted definitions. A more efficient monadic implementation of memory [43] would avoid these issues, which we leave for future work.

3.5 Related work

I have mechanised a simple program logic, including a proof of soundness, on top of the WebAssembly mechanisation described in this chapter. The program logic was designed in collaboration with Petar Maksimović, Neel Krishnaswami, and Philippa Gardner, and more details can be found in *A Program Logic for First-Order Encapsulated WebAssembly* (ECOOP 2019) [15].

The direct mechanisation of a full programming language semantics within a theorem prover is almost never attempted, due to the effort involved, not only in mechanising

the definitions, but in building any meaningful proof on top. Because WebAssembly was deliberately designed to be small, simple, and amenable to formalisation, mechanising the full semantics (with the above caveats about numerics and decoding) is more easily achievable.

It is usual for works handling a more complicated language to identify and mechanise a large interesting language core. Often, compromises are made in a mechanised semantics where some corner of the original language semantics is ambiguously defined, or complicated enough (and irrelevant to how the model will be used) that it is not worth the effort to reproduce faithfully.

JSCert [44] mechanises a semantics, and verifies an interpreter, for a large subset of JavaScript in Coq, handling all core constructs but leaving out so-called "library objects" such as `Array` and `Number` which define collections of standard library functions for their respective types. Another JavaScript semantics, λ_{JS} [45], is defined by translating JavaScript to a core calculus for which a formal semantics is specified in Coq, a process called *elaboration*.

Norrish [46] presents a mechanisation of a fragment of C in HOL. Jinja [47] is a "Java-like" mechanised semantics in Isabelle described as a "compromise between the realism of the language and the tractability and clarity of its formal semantics". Lee et al. [48] present a mechanisation in Twelf [49] of an "internal language" with "equivalent expressive power" to Standard ML, and define the semantics of Standard ML via elaboration. CakeML [50, 51] is a verified compiler for the CakeML language, which is based on a large fragment of Standard ML (minus functors), and is given a mechanised formal semantics written in HOL4. OCaml Light [52] is a mechanised semantics in HOL4 of a core subset of OCaml. The type systems of these languages are significantly more complicated than that of WebAssembly, and each of these works provide a proof of type safety for their respective models.

CompCert [53, 54], a compiler written in Coq, pre-processes a provided C source program into a core "CompCert C" language. The semantics of CompCert C itself is mechanised, as are subsequent compilation steps and associated correctness proofs, but the elaboration from C to CompCert C is unverified. Importantly, CompCert makes some narrowing assumptions about undefined behaviour in C and does not attempt to capture all outcomes allowed by the specification. This is acceptable for verifying compilation correctness, but means the model cannot be used for some other purposes.

Cerberus [55] is a model of (a fragment of) C defined purely by elaboration into a core calculus, an implementation of which is written in the Lem specification language [56]. It has been used for investigating corners of the language which are specified so ambiguously that any precise semantics would be a step forward, and for bounded model checking of small concurrent programs [57].

A way of making mechanisation and verification more tractable is to use a lightweight tool which does not have the full capabilities of a theorem prover. Such models often straddle the line between mechanisation and reference implementation since they cannot easily be used for general purpose proof.

The K framework [58], in which all semantics must be expressed as a term rewriting system, has been used to create a number of "lightweight models" for (fragments of) languages such as C [59], Java [60], JavaScript [61], and PHP [62]. An executable interpreter and limited program verification tools [63] can be automatically generated from a defined model. However, proofs over the language semantics itself (such as type soundness) are not directly expressible.

Santos et al. [64] present a symbolic evaluation tool for JavaScript programs, where the semantics of JavaScript is defined in terms of an elaboration to a simpler intermediate language ("JSIL"). The tool is later generalised to support a more generic framework for multiple languages [65]. Again, this approach is geared towards program verification and does not aim to support proofs of arbitrary language-level properties.

3.6 Evaluation and future work

The fact that the mechanisation of the draft paper semantics could be performed almost frictionlessly speaks to the precision of WebAssembly's official formalisation. Areas where the mechanisation and paper semantics needed to diverge arose exclusively from small mismatches between the metatheory of Isabelle/HOL and the implicit metatheory of the paper specification. Often, mechanisations of an existing language specification explicitly or implicitly strive for "eyeball closeness" [44], where lines of mechanisation can be closely related to lines of text in the original specification. This serves to build confidence that the mechanisation is really equivalent to the original informal specification. Because the WebAssembly specification is already fully formalised, it is easy to see where the mechanised definitions correspond to definitions in the paper semantics. However, the *order* of definitions in the mechanisation diverge in some ways from their equivalents in the paper semantics. Some of this divergence could in principle be rectified: the bulk of the mechanisation was performed while the paper semantics was still going through heavy reformatting in preparation for an official release through the W3C, and the paper specification underwent several purely editorial reorderings of sections which have not yet been fully replicated in the mechanisation. Other divergences are unavoidable due to metatheoretical differences. For example, definitions in Isabelle/HOL must be defined before they are used, and mutually recursive definitions must be explicitly defined in a single compound declaration. However in the paper specification such definitions may be split up and referenced out-of-order. An example would be the mutually recursive definitions of administrative instruction validity and local validity (§2.7).

Because the mechanisation was carried out while the specification was still in draft form, several errors were discovered and fixed before the first official release of the W3C specification (§3.1.4). Seeing this success, some members of the WebAssembly Working Group have expressed an interest in having future extensions to the language mechanised as they are developed. This is more feasible for WebAssembly than with many other languages, as Web language standardisation moves at a notoriously slow pace. Moreover, since the final stage of feature standardisation already involves writing a paper formalisation, mechanisation can both directly inform and be informed by this process. Keeping the mechanisation up to date would still be a significant commitment of time and effort, especially if associated proofs and artefacts such as the type soundness proof and verified interpreter are also to be kept updated. The mechanisation of individual post-MVP proposals would be an interesting source of small projects, and it would be worthwhile to investigate whether the mechanisation could keep pace with the evolving standard through a system of open-source contributions.

Chapter 4

CT-Wasm

CT-Wasm is a proposed type system extension to WebAssembly, designed to support the dissemination of cryptographic code that verifiably follows *information flow* and *constant-time* best practices.

CT-Wasm was originally developed and informally described by John Renner, Sunjay Cauligi, and Deian Stefan [66]. I later formalised the proposal as an extension of the existing WebAssembly semantics, mechanised the CT-Wasm extension on top of my existing mechanisation (Chapter 3), and proved a number of key security properties. The text and figures of this section are drawn partially from our previously published paper describing these contributions [10].

4.1 Background

4.1.1 Side channels and constant time

When implementing a cryptographic algorithm, functional correctness alone is not sufficient. It is also important to ensure properties about information flow that take into account the existence of *side channels*—ways in which information can be leaked as side-effects of the computation process. For example, the duration of the computation itself can be a side channel, since an attacker could compare different executions to infer which program paths were exercised, and work backwards to determine information about secret keys and messages. Naive implementations of cryptographic algorithms often leak the very information they are designed to protect via timing side channels. Kocher [67], for example, shows how a textbook implementation of RSA can be abused by an attacker to leak secret key bits. Similar key-recovery attacks were later demonstrated on real implementations (e.g., RSA [68] and AES [69, 70]). As a result, crypto-engineering best practices have shifted to mitigate such timing vulnerabilities. Many modern cryptographic algorithms are even designed with such concerns from the start [71, 72, 73].

The prevailing approach for protecting crypto implementations against timing attacks

is to attempt to ensure that the code runs in "constant time". An implementation is said to be *constant-time* if its execution time is not dependent on sensitive data, referred to as *secret* values (e.g., secret keys or messages). Constant-time implementations ensure that an attacker observing their execution behaviours cannot deduce any secret values. Though the precise capabilities of attackers vary—e.g., an attacker co-located with a victim has more capabilities than a remote attacker—most secure crypto implementations follow a conservative *constant-time programming paradigm* that altogether avoids variable-time operations, control flow, and memory access patterns that depend on secrets [74, 75].

Verifying the constant-time property (or detecting its absence) for a given implementation is considered one of the most important verification problems in cryptography [76, 77, 78, 79, 80, 81, 82, 83]. To facilitate formal reasoning, verification of the constant-time property is typically abstracted to verifying invariance of attacker knowledge under a chosen *leakage model* [84], defined over a small-step semantics for a given language. This is the approach we will be using to verify properties about CT-Wasm. A leakage model associates either a program state, or a program reduction step, with an *observation*, an abstract representation of an attacker's knowledge. For each construct of the language, the leakage model encodes what information is revealed (to an attacker) by its execution. For example, the leakage model for branching operations such as `if` or `while` leaks all values associated with the branch condition, to represent that an attacker may use timing knowledge to reason about which branch was taken [76]. Proving that a given program enjoys the constant-time property can then be abstracted as a proof that the leakage accumulated over the course of the program's execution is invariant with respect to the values of secret inputs. In general, the leakage model of a system must encompass the behaviour of hardware and compiler optimizations across all different platforms. For example, for C, operators such as division and modulus, on some architectures, are compiled to instruction sequences that have value-dependent timing. A conservative leakage model must accordingly encode these operators as leaking the values of their operands [76].

There is an unavoidable disconnect between the abstraction of a leakage model and the concrete actions of real-world compilers and architectures. Even the stringent leakage models described above are unlikely to capture all concrete timing side channels. For example, most leakage models do not take into account the timing effects of speculative execution (e.g. Spectre [85]). Still, code formally proven to satisfy a leakage-model based constant-time property is significantly more secure than unverified code, and implementations designed with such models in mind have proven useful in practice. For example, the HACL* library [80] has been adopted by Firefox [86], while Fiat [83] has been adopted by Chrome.

4.1.2 Crypto on the Web

Writing code that does not leak information via side channels is daunting even with complete control over the execution environment, but in recent years an even more challenging environment has emerged—that of *in-browser cryptography*—the implementation of cryptographic algorithms in a user's browser using JavaScript. As discussed, for decades JavaScript was the only programming language natively supported on the Web. This has created many scenarios where JavaScript was the *only* frictionless option for a Web developer looking to implement some behaviour or feature directly in a Web page, and in many cases it is also the convenient choice for any accompanying server-side development. Modern JavaScript runtimes are extremely complex software systems, incorporating caching, multi-tiered just-in-time (JIT) compilation, speculative optimisation, and garbage collection (GC) techniques that almost inherently expose timing side-channels [87, 88, 89]. Even worse, much of the JavaScript cryptography used in the wild is implemented by "unskilled cryptographers" [90] who do not account for even the most basic timing side channels.

Existing research on cryptography verification is difficult to apply to JavaScript. Defining a leakage model for JavaScript is extremely difficult, due to the complexity of the language. Even if we restrict ourselves to semantically more simple subsets of JavaScript (e.g., asm.js [4] or defensive JavaScript [91]), the leakage model is still dependent on the behaviour of the complex JavaScript runtime.

Despite these theoretical shortcomings, JavaScript crypto libraries remain overwhelmingly popular [92, 93, 94, 95, 96, 97]. JavaScript APIs for native crypto libraries have been standardised with the specific goal of reducing the prevalence of bespoke JavaScript cryptography [98], but unfortunately have suffered from significant divergence between platforms. For example the Web Crypto [98] and the Node.js crypto [99] APIs (available to browser and server-side JavaScript respectively) barely overlap, undercutting a major motivation for using JavaScript in the first place—its cross-platform nature. Moreover, these libraries did not keep up with new trends, with slow support for primitives such as Poly1305 [72] which are widely used in modern crypto applications [3]. Due to these issues, even "secure" applications such as Signal [92] and Cryptocat [93] routinely default to JavaScript cryptography.

4.1.3 WebAssembly

WebAssembly is designed to be compiled directly to machine code without the need for speculative JIT-style optimisations. This alone provides a firmer foundation for cryptography than JavaScript: WebAssembly's close-to-the-metal instructions give us more confidence in its timing characteristics than is possible when dealing with JavaScript's unpredictable optimisations. Just as importantly, WebAssembly is designed to appeal to

the same Web ecosystem-adjacent communities that currently use JavaScript. An explicit goal was to make migration from JavaScript to WebAssembly as seamless as possible, through tooling and engine support.

With CT-Wasm, we show how WebAssembly can be extended with with cryptographically meaningful types, which provide verifiable security guarantees. CT-Wasm's type system draws from previous assembly language type systems that enforce constant-time [100]. Our system, however, is explicitly designed and validated for the in-browser crypto use case. We ensure that our type system extensions maintain WebAssembly's extremely fast type checking, so that our properties can be checked as part of the single linear type checking pass. In a similar way to Safe Haskell [101], we extend WebAssembly's module system to further allow developers to specify if a particular import is trusted or untrusted. In combination with our other type system extensions, this allows developers to safely delineate the boundary between their own code and third-party, untrusted code. We also show that popular algorithms such as Poly1305, among many other cryptographic algorithms, can be securely implemented and typed in CT-Wasm.

4.2 CT-Wasm overview

Constant-Time WebAssembly pursues four main design goals. First, CT-Wasm must identify and reject programs which exhibit information flow or constant-time violations. This is the most fundamental aim of the extension: certain data will be designated by the programmer as sensitive, and CT-Wasm must enforce that this data is not used in a way which compromises our leakage model.

Second, CT-Wasm should be expressive enough to implement real-world crypto algorithms. This requirement is self-explanatory; there is no point in defining a system which is too restrictive to be used by real programs.

Third, since WebAssembly and most crypto algorithms are designed with performance in mind, CT-Wasm must not incur significant overhead, either from validation or execution. Overall page load time is considered one of—if not *the*—key Web site performance metrics [102, 103], so requiring the Web client to conduct expensive analyses of loaded code before execution would be infeasible.

Fourth, CT-Wasm should maintain backwards compatibility (and, as far as possible, interoperability) with legacy WebAssembly code. This property is necessary to allow CT-Wasm to be proposed for adoption by the language standard.

CT-Wasm's restrictions are implemented through extensions to WebAssembly's type system. We create a type-level distinction between secret and public data. We augment WebAssembly's four base types with explicit types for secret integers **s32** and **s64**. Floating-point types are always considered public, since the vast majority of floating-point operations are variable-time and therefore vulnerable to timing attacks [104, 105, 106].

We also extend type checking to ensure that such secret data cannot be leaked either directly (e.g., by writing secret values to public memory) or indirectly (e.g., via control flow and memory access patterns), by imposing secure information flow and constant-time disciplines on code that handles secrets (see §4.3). Finally, we make a distinction between untrusted code, which must obey these information flow restrictions, and trusted code which may break them by explicitly "declassifying" secret data to public. Consider, for example, a function which takes a secret message and encrypts it, returning the encrypted data. In this example, this function is untrusted, and can only (transitively) call other untrusted functions. Our type system forces the encrypted data to be typed as secret due to an information flow dependency on the value of the original message. At the top level, however, the program must make the deliberate decision to declassify the encrypted data in order to send it, relying on the mathematical/computational security of the encryption algorithm rather than an information flow property. This code will be marked as trusted, to explicitly record a place where the properties of CT-Wasm can be broken. It is up to the programmer to ensure that such code correctly manipulates the results of untrusted functions in a way that keeps secret data secure.

CT-Wasm's restrictions are enforced statically in a single pass, as an extension to WebAssembly's regular validation procedure. This means that the restrictions on the flow of secret data must be very coarse-grained. Our type system is more restrictive than more traditional low-level information flow control type systems (e.g., JIF's [107, 108, 100, 76] or that of FlowCaml [109]). We are still able to successfully implement a wide selection of cryptographic algorithms under these restrictions, however. As it turns out, cryptography developers already impose these restrictions upon their code as "best practice" [74, 75]: our type checker effectively ensures that untrusted code respects these (previously) self-imposed limitations.

4.2.1 Attacker model

We assume that the attacker can **(1)** supply arbitrary untrusted CT-Wasm functions and manipulate the target into executing them on secret data, and **(2)** observe the runtime behaviour of the target according to the leakage model we define below.

4.2.1.1 Leakage model

Each step of program execution is associated with an observation, which represents the information revealed to an attacker by that execution step. We assume the following leakage:

- The attacker knows *which* operation is being executed (e.g. the attacker can distinguish between a multiplication and a division, even if the operands are the same).

- Operands to conditional control flow (i.e. **if**, **br_if**, **br_table**, **call_indirect**) are leaked, which over-approximates observable timing differences between different execution paths.

- Memory access operations (i.e. **load**/**store**) leak their index operands, to represent that these operands can be revealed by cache timing. The current memory length is also leaked by all memory operations (including **size**/**grow**), over-approximating not only the conditional bounds-checking behaviour which may reveal the size, but also any OS-level timing behaviour which may be assocated with memory allocation.

- Some binary operations are explicitly marked as *unsafe*, representing that their execution time may be input-dependent. Such operations leak the values of their operands. The typing rules (Fig. 4.2) use this list to decide whether an arithmetic operation can be carried out on secret operands. All floating point operations are assumed unsafe. It is common to assume that integer division and modulus are unsafe, and these operations are rarely used in cryptographic algorithms. However even multiplication and bitshift can have timing vulnerabilities on certain CPUs [75]. The specific choice of safe binary operations must be made pragmatically. In practical implementations of CT-Wasm, we forbid division and modulus and allow multiplication and bitshift, in line with the assumptions made by Zinzindohoué et al. [80]. The type system, proofs, and leakage model described in this chapter can in principle be adjusted to allow/forbid different operations (with knock-on effects to the practicality of using the system).

- When a trusted host function is called, it is assumed that the entire runtime state is leaked.

Note that, as discussed, this leakage model does not cover speculative execution attacks. Leakage models which cover these attacks would be too leaky for any coarse-grained type system to offer meaningful protection [110]. It is assumed that vulnerabilities due to speculative execution must be mitigated through some orthogonal mechanism such as process isolation [111]. We discuss the limitations of our approach in more detail in §4.3.4.

4.2.2 CT-Wasm formally

CT-Wasm requires only small changes to the language's AST and reduction rules. These are shown in Fig. 4.1. The integer types are augmented with a secrecy parameter, and the existing i32 and i64 types are defined to represent public integers specifically, with the new types s32 and s64 representing secret integers. The definitions of most existing type-annotated instructions such as *t.binop* are not altered, but they will inherit this extended type definition in their annotations. The select operator is extended with an

explicit secrecy annotation to simplify the typing rules. Two new conversion operators are added between secret and public types. Their dynamic behaviours are trivial, and their presence does not imply any concrete runtime overhead, but the use of **declassify** will be restricted by the type system, as seen shortly. Functions are annotated with a trust parameter, which will determine whether they are allowed to **declassify** or call another trusted function. Memories are annotated with a secrecy parameter. We keep track of secrecy on a coarse-grained memory-by-memory basis because a more fine-grained approach would require dynamic checks to ensure that a secret value is not stored in a public "location". With our approach, a store may only target a public memory if its argument can be statically typed as public.

The only place where a trust/secrecy annotation must be dynamically checked is with the **call_indirect** instruction. Although the reduction rule for **call_indirect** does not change (see §2.4.5), the addition of a trust component to each function type means the rule's dynamic type check is the one place in the semantics where the trust/secrecy annotation must be dynamically checked.

$$\text{(value types) } t ::= \mathsf{i32}' \; sec \mid \mathsf{i64}' \; sec \mid \mathsf{f32} \mid \mathsf{f64}$$

$$\begin{array}{llll} \mathsf{sec}\ (\mathsf{i}N'\ sec) & \triangleq & sec & \qquad \mathsf{i}N \triangleq \mathsf{i}N'\ \mathsf{public} \\ \mathsf{sec}\ \mathsf{f}N & \triangleq & \mathsf{public} & \qquad \mathsf{s}N \triangleq \mathsf{i}N'\ \mathsf{secret} \end{array}$$

$$\begin{array}{lll} \text{(trust)} & tr ::= \mathsf{trusted} \mid \mathsf{untrusted} & cvtops ::= \ldots \mathsf{classify} \mid \mathsf{declassify} \\ \text{(secrecy)} & sec ::= \mathsf{public} \mid \mathsf{secret} & e ::= \ldots \mathsf{select}\ sec \mid \ldots \end{array}$$

$$\text{(instance) } inst ::= \{\ \text{types} :: (tr, ft)^*,\ \text{ifuncs} :: i^*,\ \text{itabs} :: i^*,\ \text{imems} :: i^*,\ \ \text{iglobs} :: i^*\ \}$$

$$\begin{array}{rl} \text{(function closure)} & cl ::= \mathsf{native}\ \{\ \text{inst} :: inst, \text{type} :: (tr, ft), \text{locals} :: t^*, \text{body} :: e^*\} \mid \\ & \qquad \mathsf{host}\ \{\ \text{type} :: (tr, ft), \text{host} :: hostfunc\} \end{array}$$

$$\text{(memory) } mem ::= \{\ \mathsf{sec} :: sec,\ \ \text{buffer} :: byte,\ \ \text{max} :: nat\ \}$$

$$\begin{array}{rcl} S; F;\ (\mathsf{s}N.\mathsf{const}\ k)\ t_2.\mathsf{declassify}\ t_1 & \hookrightarrow & S; F;\ (\mathsf{i}N.\mathsf{const}\ k) \\ S; F;\ (\mathsf{i}N.\mathsf{const}\ k)\ t_2.\mathsf{classify}\ t_1 & \hookrightarrow & S; F;\ (\mathsf{s}N.\mathsf{const}\ k) \end{array}$$

Figure 4.1: CT-Wasm modifications to the runtime AST and reduction rules (highlighted)

4.2.2.1 Type system

The type system of CT-Wasm, defined in Fig. 4.2, is where the vast majority of CT-Wasm's new restrictions are defined. Essentially it is a very naive, coarse-grained information flow type system (a variant of the Volpano-Irvine-Smith system [112]) with a **declassify** escape hatch allowed only in functions marked trusted. To enforce a constant-time discipline,

secret values may not be used as arguments to expressions with timing-sensitive behaviour. A secret value may not be used as the argument to a conditional (**if**, **br_if**, **br_table**, **call_indirect**), or as an index into memory (**load**, **store**). Note that while the typing rules for **if**, **br_if**, and **br_table** are syntactically unchanged and thus not included in Fig. 4.2, they are implicitly restricted by our redefinition of the i32 type to refer specifically to public integers, as shown in Fig. 4.1. Only a restricted subset of binary operations are allowed on secret values, enforced by the is_safe condition, as some operations such as division are known to be variable-time on common architectures. The precise definition of is_safe will be discussed as part of the leakage model (§4.3).

There is no subtyping between secret and public value types. Instead, explicit conversion instructions must be inserted. This means that the typing rule for (e.g.) *binop* automatically enforces our information flow requirements. If the input type is secret, the output type must be as well. In addition, secret and public arguments cannot be mixed. Instead, the public input must be explicitly classified, ensuring the output is correctly marked as secret.

It is a type error for **declassify** to occur in an untrusted function, and the rule for **call** statically ensures that untrusted functions can only transitively call other untrusted functions. The **call_indirect** instruction cannot be fully checked as part of the initial type checking pass. Instead, as noted earlier, it must perform a runtime check that the called function respects its type annotation. The annotation may only denote a trusted function if the current function is also trusted.

Finally, there are the restrictions on secret and public memory. As mentioned above, the index used to access memory is always considered public, even if the accessed memory is secret. This is because of cache timing effects which we assume will reveal any memory access patterns. Note that this does not model Spectre, since our model only leaks addresses which are actually accessed as part of the execution trace, while Spectre allows loads and stores which are only speculatively executed to cause leakage. A loaded value will be considered secret if the memory it is loaded from is secret, and a secret value can only be stored into secret memory. The size of memory is also always considered public (since an attacker could observe bounds checking errors).

4.3 Security property formalisation

The definitions above were mechanised as an extension to the original Isabelle mechanisations of the type system and reduction rules (§3). The verified type checker was also updated to check CT-Wasm programs. All code can be found in the supplemental material [16].

$$\text{(memory type)}\quad mt ::= (limit, \mathit{sec})$$

$$\text{(contexts)}\quad C ::= \left\{ \begin{array}{l} \mathsf{trust}\ tr,\ \mathsf{func}\ (tr, ft)^*,\ \mathsf{global}\ gt^*,\ \mathsf{table}\ tt^*, \\ \mathsf{memory}\ mt^*,\ \mathsf{local}\ t^*,\ \mathsf{label}\ (t^*)^*,\ \mathsf{return}\ (t^*)^? \end{array} \right\}$$

$$tr \succ_{\mathbf{tr}} tr' \triangleq (tr = tr') \vee (tr = \mathsf{trusted} \wedge tr' = \mathsf{untrusted})$$

$$\frac{\mathsf{sec}\ t = \mathsf{secret} \implies \mathsf{is\_safe}(binop_t)}{C \vdash t.binop_t : t\ t \to t}$$

$$\frac{\mathsf{sec}\ t = \mathit{sec}}{C \vdash t.testop : t \to (\mathsf{i32}'\ \mathit{sec})} \qquad \frac{\mathsf{sec}\ t = \mathit{sec}}{C \vdash t.relop : t\ t \to (\mathsf{i32}'\ \mathit{sec})}$$

$$\frac{\mathsf{sec}\ t_1 = \mathsf{sec}\ t_2}{C \vdash t_2.\mathbf{convert}_{(t_1, sx^?)} : t_1 \to t_2} \qquad \frac{\mathsf{sec}\ t_1 = \mathsf{sec}\ t_2}{C \vdash t_2.\mathbf{reinterpret}_{(t_1, sx^?)} : t_1 \to t_2}$$

$$\frac{(t_1 = \mathsf{i}n'\ \mathsf{secret} \wedge t_2 = \mathsf{i}n'\ \mathsf{public})}{C \vdash t_1.\mathbf{classify}\ t_2 : t_2 \to t_1} \qquad \frac{C.\mathrm{trust} = \mathsf{trusted} \qquad (t_1 = \mathsf{i}n'\ \mathsf{public} \wedge t_2 = \mathsf{i}n'\ \mathsf{secret})}{C \vdash t_1.\mathbf{declassify}\ t_2 : t_2 \to t_1}$$

$$\frac{\mathit{sec} = \mathsf{secret} \longrightarrow\ \mathsf{sec}\ t = \mathsf{secret}}{C \vdash \mathbf{select}\ \mathit{sec} : t\ t\ (\mathsf{i32}'\ \mathit{sec}) \to t} \qquad \frac{C.\mathrm{trust} = tr \quad C.\mathrm{func}[i] = (tr', ft) \quad tr \succ_{\mathbf{tr}} tr'}{C \vdash \mathbf{call}\ i : ft}$$

$$\frac{ft = t_1^* \to t_2^* \qquad C.\mathrm{trust} = tr \qquad tr \succ_{\mathbf{tr}} tr' \qquad C_{\mathsf{table}} = n}{C \vdash \mathbf{call\_indirect}\ (tr', ft) : t_1^*\ \mathsf{i32} \to t_2^*}$$

$$\frac{C.\mathrm{memory}[0] = (lim, \mathit{sec}) \qquad \mathsf{sec}\ t = \mathit{sec} \qquad 2^a \leqslant |t|}{C \vdash t.\mathbf{load}\ -\ a\ o : \mathsf{i32} \to t}$$

$$\frac{C.\mathrm{memory}[0] = (lim, \mathit{sec}) \qquad \mathsf{sec}\ t = \mathit{sec} \qquad 2^a \leqslant |t|}{C \vdash t.\mathbf{store}\ -\ a\ o : \mathsf{i32}\ t \to \epsilon}$$

$$\frac{C.\mathrm{memory}[0] = (lim, \mathit{sec}) \qquad \mathsf{sec}\ t = \mathit{sec} \qquad 2^a \leqslant |pt| < (m/8) \qquad t = \mathsf{i}m'\ \mathit{sec}}{C \vdash t.\mathbf{load}\ (pt, sx)\ a\ o : \mathsf{i32} \to t}$$

$$\frac{C.\mathrm{memory}[0] = (lim, \mathit{sec}) \qquad \mathsf{sec}\ t = \mathit{sec} \qquad 2^a \leqslant |pt| < |t| \qquad t = \mathsf{i}m'\ \mathit{sec}}{C \vdash t.\mathbf{store}\ pt\ a\ o : \mathsf{i32}\ t \to \epsilon}$$

Figure 4.2: Typing rules of CT-Wasm where they differ (highlighted) from the rules of §2.6.

4.3.1 Soundness

Before conducting the constant-time proof, the *preservation* property (§2.7) must be updated to also prove that the trust level of the code is preserved by execution. That is, if a code fragment reduces in an untrusted context, the reduct will only contain instructions which are allowed in an untrusted context.

The top-level configuration typing relation is updated to also assign a trust level tr to typed code. Local validity is then updated to require that the code must be typed at that trust level.

$$\frac{\vdash_{\mathsf{s}} S : \mathsf{ok} \quad S; tr; \epsilon \vdash_{\mathsf{loc}} F; e^* : (tr,\ t^*)}{\vdash_{\mathsf{c}} S; F; e^* : (tr,\ t^*)}$$

$$\frac{S \vdash_{\mathsf{f}} F : C \qquad S; C[\text{trust} := tr, \text{return} := (t_r^*)^?] \vdash e^* : \epsilon \to t^*}{S; tr; (t_r^*)^? \vdash_{\mathsf{loc}} F; e^* : t^*}$$

This new preservation property is proven by extending the rule induction of §3.1.2 to also prove that the trust level is either preserved, or, in the case of function call, respects the simple trust lattice $\succ_{\mathbf{tr}}$ defined in Fig. 4.2.

4.3.2 Leakage model

Formally, there are a number of equivalent ways to represent attacker observations due to leakage. We choose to augment the WebAssembly reduction rule with actions (a) which record the relevant inputs to the executed instruction. The execution of a WebAssembly program gives rise to an associated trace of events. Then, an *observational equivalence* relation ($\approx_{\mathsf{a}}$) is defined between actions, which captures the observational power of the attacker as discussed above.

To formalise the previously described leakage model (§4.2.1.1), each reduction rule (§2.4) is associated with an action which records the operation executed, and relevant stack operands, static parameters, and program state (such as memory length in the case of **load**/**store**). For example, the reduction rules for binary operations are extended as follows:

$$(t.\mathbf{const}\ j)(t.\mathbf{const}\ k)(t.binop_t) \hookrightarrow^a (t.\mathbf{const}\ l) \quad \text{where} \quad binop_t(j,k) = l,\ a = \mathsf{act\_binop\_succ}(j,k,binop_t)$$

$$(t.\mathbf{const}\ j)(t.\mathbf{const}\ k)(t.binop_t) \hookrightarrow^a (\mathbf{trap}) \quad \text{where} \quad binop_t(j,k) = \bot,\ a = \mathsf{act\_binop\_fail}(j,k,binop_t)$$

The $\sim_a$ relation between these actions is defined as follows:

$$\mathsf{act\_binop\_succ}(j,k,op) \approx_{\mathsf{a}} \mathsf{act\_binop\_succ}(j',k',op') \triangleq op = op' \wedge (\mathrm{is\_safe}(op) \vee (j = j' \wedge k = k'))$$

($\mathsf{act\_binop\_fail}$ is analogous)

Value, global, memory, and frame indistinguishability

$$t_1.\mathbf{const}\ k_1 \quad \sim_{\mathsf{v}} \quad t_2.\mathbf{const}\ k_2 \quad \triangleq \quad t_1 = t_2 \wedge (k_1 = k_2 \vee \mathsf{sec}\ t_1 = \mathsf{sec}\ t_2 = \mathsf{secret})$$

$$glob_1 \quad \sim_{\mathsf{g}} \quad glob_2 \quad \triangleq \quad glob_1.\mathrm{mut} = glob_2.\mathrm{mut} \wedge glob_1.\mathrm{val} \sim_{\mathsf{v}} glob_2.\mathrm{val}$$

$$mem_1 \quad \sim_{\mathsf{m}} \quad mem_2 \quad \triangleq \quad \left(\begin{array}{l} mem_1.\mathrm{sec} = mem_2.\mathrm{sec} = \mathsf{secret} \\ \wedge\ |mem_1.\mathrm{buffer}| = |mem_2.\mathrm{buffer}| \end{array}\right) \vee (mem_1 = mem_2)$$

$$F_1 \quad \sim_{\mathsf{f}} \quad F_2 \quad \triangleq \quad F_1.\mathrm{inst} = F_2.\mathrm{inst} \wedge \ \mathrm{list\_all2}\ (\sim_{\mathsf{v}})\ F_1.\mathrm{locs}\ F_2.\mathrm{locs}$$

$$\mathrm{list\_all2} :: (a \Rightarrow b \Rightarrow \mathit{bool}) \Rightarrow a\ \mathit{list} \Rightarrow b\ \mathit{list} \Rightarrow \mathit{bool}$$

Expression indistinguishability

$$e_1 \sim_{\mathsf{e}} e_2 \triangleq \begin{cases} (e_a \sim_{\mathsf{v}} e_b) & \text{if} \quad \begin{array}{l} e_1 = t_1.\mathbf{const}\ k_1 \\ e_2 = t_2.\mathbf{const}\ k_2 \end{array} \\ (e_a \sim_{\mathsf{e}} e_b)^n & \text{if} \quad \begin{array}{l} e_1 = \mathbf{block}\ \mathit{ft}\ e_a^n\ \mathbf{end} \\ e_2 = \mathbf{block}\ \mathit{ft}\ e_b^n\ \mathbf{end} \end{array} \ \text{or} \ \begin{array}{l} e_1 = \mathbf{loop}\ \mathit{ft}\ e_a^n\ \mathbf{end} \\ e_2 = \mathbf{loop}\ \mathit{ft}\ e_b^n\ \mathbf{end} \end{array} \\ (e_a \sim_{\mathsf{e}} e_b)^n \\ \wedge\ (e_c \sim_{\mathsf{e}} e_d)^m & \text{if} \quad \begin{array}{l} e_1 = \mathbf{if}\ \mathit{ft}\ e_a^n\ \mathbf{else}\ e_c^m\ \mathbf{end} \\ e_2 = \mathbf{if}\ \mathit{ft}\ e_b^n\ \mathbf{else}\ e_d^m\ \mathbf{end} \end{array} \ \text{or} \ \begin{array}{l} e_1 = \mathbf{label}_n\{e_a^n\}\ e_c^m\ \mathbf{end} \\ e_2 = \mathbf{label}_n\{e_b^n\}\ e_d^m\ \mathbf{end} \end{array} \\ (F_a \sim_{\mathsf{f}} F_b) \\ \wedge\ (e_a \sim_{\mathsf{e}} e_b)^m & \text{if} \quad \begin{array}{l} e_1 = \mathbf{frame}_n\{F_a\}\ e_a^m\ \mathbf{end} \\ e_2 = \mathbf{frame}_n\{F_b\}\ e_b^m\ \mathbf{end} \end{array} \\ e_1 = e_2 & \text{otherwise} \end{cases}$$

Store indistinguishability

$$S_1 \ \sim_{\mathsf{s}} S_2 \ \triangleq \ \begin{array}{l} S_1.\mathrm{funcs} = S_2.\mathrm{funcs} \\ \wedge\ S_1.\mathrm{tabs} = S_2.\mathrm{tabs} \\ \wedge\ \mathrm{list\_all2}\ (\sim_{\mathsf{g}})\ S_1.\mathrm{globs}\ S_2.\mathrm{globs} \\ \wedge\ \mathrm{list\_all2}\ (\sim_{\mathsf{m}})\ S_1.\mathrm{mems}\ S_2.\mathrm{mems} \end{array}$$

Configuration indistinguishability

$$S_1; F_1^*; e_1^* \ \sim_{\mathsf{c}} \ S_2; V_2^*; e_2^* \triangleq (S_1 \sim_{\mathsf{s}} S_2) \wedge (F_1 \sim_{\mathsf{f}} F_2) \wedge (e_1 \sim_{\mathsf{e}} e_2)^*$$

Figure 4.3: Definition of secret indistinguishability.

4.3.3 Constant-time

We formalise constant-timedness in the standard way, as the property that well-typed untrusted configurations which vary only in their secret state must produce observationally equivalent (according to $\approx_{\mathsf{a}}$) traces. The first step is to define a *secret indistinguishability* relation ($\sim_{\mathsf{c}}$) between configurations, which formalises what it means for a pair of configurations to vary only in public state (Fig. 4.3). We can prove that two indistinguishable configurations must have the same type.

Lemma 4.1 (`config_indistinguishable_imp_config_typing`)

Assuming (1) $\vdash_{\mathsf{c}} \mathit{Cfg} : (\mathsf{untrusted}, t^*)$
(2) $\mathit{Cfg} \sim_{\mathsf{c}} \mathit{Cfg}'$

we have (2) $\vdash_{\mathsf{c}} \mathit{Cfg}' : (\mathsf{untrusted}, t^*)$

Proof. Follows directly from the definitions.

We can now state and prove a one-step *unwinding lemma* that will imply our desired constant-time property.

Lemma 4.2 (`config_indistinguishable_imp_reduce`)

Assuming (1) $\vdash_{\mathsf{c}} \mathit{Cfg} : (\mathsf{untrusted}, t^*)$
(2) $\mathit{Cfg} \hookrightarrow^{a} \mathit{Cfg}_{\mathsf{a}}$
(3) $\mathit{Cfg} \sim_{\mathsf{c}} \mathit{Cfg}'$

there exist $\mathit{Cfg}'_{\mathsf{a}}$, a', *such that* (4) $\mathit{Cfg}' \hookrightarrow^{a'} \mathit{Cfg}'_{\mathsf{a}}$
(5) $\mathit{Cfg}_{\mathsf{a}} \sim_{\mathsf{c}} \mathit{Cfg}'_{\mathsf{a}}$
(6) $a \approx_{a} a'$

Proof. By induction on the definition of reduction.

This lemma states that, given a well-typed untrusted configuration which reduces one step and emits an action a, any other indistinguishable configuration can also reduce one step in a way that preserves indistinguishability, while emitting an action a' which is observationally equivalent to a. The lemma in fact implies a stronger property than just constant-time, since (combined with the type preservation property), it guarantees that all indistinguishable untrusted configurations reduce in precise lock-step with each other, not just that they produce indistinguishable traces. This is the *self-isomorphism* property of Popescu et al. [113], which is known to imply a number of weaker non-interference properties.

In any case, we can formalise the constant-time property and show that it is implied by this lemma. First, we formalise the notion of an action trace as the potentially infinite

sequence of actions associated with a configuration according to the coinductive closure of the reduction relation.

$$\frac{\nexists Cfg'\ a.\ Cfg \hookrightarrow^{a} Cfg'}{Cfg \vDash_{\mathsf{trace}} \epsilon}$$

$$\frac{Cfg \hookrightarrow^{a} Cfg' \qquad Cfg' \vDash_{\mathsf{trace}} tr}{Cfg \vDash_{\mathsf{trace}} (a : tr)}$$

Note that $\vDash_{\mathsf{trace}}$ is defined coinductively rather than inductively, as indicated by the doubled line. We then define observational equivalence between traces ($\approx_{\mathsf{tr}}$) via corecursive pairwise comparison by $\approx_{\mathsf{a}}$.

$$\frac{}{\epsilon \approx_{\mathsf{tr}} \epsilon}$$

$$\frac{a \approx_{\mathsf{a}} a' \qquad tr \approx_{\mathsf{tr}} tr'}{(a : tr) \approx_{\mathsf{tr}} (a' : tr')}$$

Finally, we can lift the $\approx_{\mathsf{tr}}$ relation to a relation ($\approx_{\mathsf{trs}}$) between *sets* of traces in the standard way, and define the constant-time property with respect to a given configuration *Cfg*.

$$trs_1 \approx_{\mathsf{trs}} trs_1 \quad \triangleq \quad \begin{array}{l}(\forall tr_1 \in trs_1.\ \exists tr_2 \in trs_2.\ tr_1 \approx_{\mathsf{tr}} tr_2) \\ \wedge\ (\forall tr_2 \in trs_2.\ \exists tr_1 \in trs_1.\ tr_2 \approx_{\mathsf{tr}} tr_1)\end{array}$$

$$\text{constant-time}(Cfg) \quad \triangleq \quad \forall Cfg'.\ Cfg \sim_{\mathsf{c}} Cfg' \Longrightarrow \{\ tr \mid Cfg \vDash_{\mathsf{trace}} tr\ \} \approx_{\mathsf{trs}} \{\ tr' \mid Cfg' \vDash_{\mathsf{trace}} tr'\ \}$$

We can now prove that well-typed untrusted configurations are constant time.

Lemma 4.3 (`config_indistinguishable_imp_reduce2`)

Assuming (1) $\vdash_c Cfg : (\mathsf{untrusted}, t^*)$
(2) $Cfg \vDash_{\mathsf{trace}} tr$
(3) $Cfg \sim_c Cfg'$
we have (4) $\exists tr'.\ Cfg' \vDash_{\mathsf{trace}} tr' \quad \wedge \quad tr \approx_{\mathsf{tr}} tr'$

Proof. By coinduction on the definition of $\vDash_{\mathsf{trace}}$,
using `config_indistinguishable_imp_reduce` and `preservation`.

Lemma 4.4 (`config_untrusted_constant_time`)
Assuming (1) $\vdash_c Cfg : (\mathsf{untrusted}, t^*)$
we have (2) constant-time(Cfg)

Proof. Follows directly from the definitions,
using `config_indistinguishable_imp_config_typing`
and `config_indistinguishable_imp_reduce2`.

4.3.4 Limitations

As previously mentioned, our formalisation of leakage fits the standard assumptions of the constant-time programming discipline but is an imperfect abstraction over the "real world". The leakage model must make assumptions about which WebAssembly-level operations have timing vulnerabilities. In a concrete implementation, the validity of these assumptions will depend on both the particular WebAssembly compiler and underlying target architecture. Our type system carries the necessary information to the implementation regarding which operations are assumed to be constant-time, but microarchitectural timing behaviour is complicated and often undocumented, so there is a limit to how confident implementations can be in the safety of their platform instruction selection. When concretely implementing CT-Wasm below, we assume multiplication is constant-time, which is true on most CPUs but would require the implementation to use inefficient workarounds during instruction selection when targetting the few CPUs for which this is not the case [75]. This per-CPU mitigation is more feasible in a Web scenario because compilation to platform assembly is performed by the Web engine directly running on the executing machine, although it is still more onerous that most implementers are likely to countenance.

Speculative execution attacks such as Spectre [85] are not modelled and must be mitigated independently. We also do not model attacks based on observing power consumption [114].

As is standard for an information flow type system, a wrapper of trusted code is necessary in order to usefully manipulate data returned by untrusted code. For example, an untrusted algorithm may be used to encrypt a message with a secret key, and the resulting ciphertext will be tainted as secret. Is the responsibility of the trusted code to choose to declassify the ciphertext (essentially, explicitly trusting in the functional correctness and computational hardness of the encryption) so that it can be sent over the wire, and the CT-Wasm type system gives no guarantees about the security of this choice.

4.4 Evaluation

The work described in this section was primarily performed by other collaborators. The experiments are reported in more detail by Watt et al. [10] but some key results are summarised here for context.

The type system of CT-Wasm is very simple and coarse-grained in comparison to many other information flow type systems. The security level of data must be explicitly fixed as either secret or public at every value use and control flow join. This coarse-grained design allows for extremely fast validation, which is essential given our previously mentioned concerns about startup time. However this approach is only acceptable if common cryptography primitives are actually expressible within the type system. To evaluate this, a variety of common cryptography primitives were successfully implemented in CT-Wasm, including the previously mentioned Poly1305 algorithm. This success was not surprising, as our type system aimed to match existing cryptographic implementation best practices.

A fork of the V8 browser engine was experimentally modified to support CT-Wasm, and the `dudect` fuzzing and statistical analysis tool [115] was used to confirm that no timing vulnerabilities were detectable through fuzzing when our CT-Wasm code was executed in this implementation. In contrast, when we implemented a textbook definition of the Salsa20 primitive partially in JavaScript, `dudect` quickly detected a minor timing leak: V8 would box any JavaScript integer outside a 31-bit signed range [116], leading to reduced performance on some inputs even if all numbers were kept in a 32-bit range.

I extended my verified type checker (§3.3.1) to check CT-Wasm's additional restrictions, although this implementation was not used as part of the experimental evaluations except to double-check the behaviour of the V8 implementation.

4.5 Related and future work

Low-level crypto DSLs Bernstein's qhasm [117] is an assembly-level language used to implement many cryptographic routines, including the core algorithms of the NaCl library. The language does not enforce a particular secure programming discipline; this remains the responsibility of the programmer.

Vale [81] and Jasmin [79] are structured assembly languages designed for high-performance cryptography, and have verification systems to prove functional correctness in addition to side-channel freedom. Vale and Jasmin both target native machine assembly, and rely upon the Dafny verification system [118]. Vale uses a flow-sensitive type system to enforce non-interference, while Jasmin uses Boogie-based verification [119]. CT-Wasm does not address functional correctness, and relies on a much simpler, but significantly faster verification procedure.

High-level crypto DSLs The HACL* [80] cryptographic library is written in constrained subsets of the F* verification language that can be compiled to C. HACL* primitives are proven functionally correct and free of side-channels (making assumptions similar to the leakage model of CT-Wasm) through individual proofs in the F* theorem prover, with some automation through SMT. The HACL* authors are also investigating WebAssembly as a compilation target [120]. FaCT [82] is a high-level language that enforces a constant-time programming discipline and targets LLVM. CAO [121, 122] and Cryptol [123] are high-level DSLs for crypto implementations which do not enforce a secure programming discipline.

All these efforts are complementary to our lower-level approach, and could benefit from targetting CT-Wasm.

Leakage models Our leakage model derives much of its legitimacy from existing work on the side-channel characteristics of low-level languages, both practical [115, 74] and theoretical [76, 100, 78]. We aim to express our top-level information flow and constant-time properties in a way that is familiar to readers of these works. Our representation of observations draws inspiration from the equivalence relation-based formalisations described by Sabelfeld and Myers [124] for timing sensitive non-interference, and used elsewhere in similar proofs [100, 125, 126].

Standardisation CT-Wasm has been proposed for official adoption into the WebAssembly specification, and is currently at an early stage of standardisation. Even if CT-Wasm types are checked by implementations, subtle choices of compiler optimisation or instruction selection may sabotage our security guarantees. A more thorough investigation is needed to determine the feasibility of securely implementing CT-Wasm in existing Web engines before further stages of standardisation can proceed.

Chapter 5

Relaxed memory

This section describes my work on the relaxed memory models of JavaScript and WebAssembly. WebAssembly's threads specification is still in progress, and I have been responsible for defining the relaxed memory model which has been incorporated into the draft specification. Many design decisions for WebAssembly's behaviour are constrained by the requirement that WebAssembly must seamlessly interoperate with JavaScript, which has its own pre-existing concurrency semantics and relaxed memory model. Therefore, WebAssembly's memory model is primarily an extension of that JavaScript memory model. The bulk of this chapter deals with the investigation and critique of JavaScript's relaxed memory model, followed by my work in extending the model to cover WebAssembly's unique features.

Several issues with the JavaScript model are discovered and corrected, with these changes officially adopted by the JavaScript standards body, and incorporated into WebAssembly's memory model in its draft threads specification. An outstanding issue regarding JavaScript's compatibility with the C++11 memory model is reported on. Then, the WebAssembly-specific extensions to JavaScript's model are described. This work was carried out in collaboration with a number of other academics and industry figures, as detailed below. The chapter draws from two previously published papers [13, 15].

5.1 Contributions

In order to ensure the WebAssembly relaxed memory model is built on as solid a foundation as possible, I worked with collaborators to identify and (where possible) correct several issues with the existing JavaScript model.

Wait/notify synchronisation The JavaScript language provides the Atomics.wait and Atomics.notify operations, which allow threads to be suspended and unsuspended. I discovered that these operations were not properly modelled in the language's formal

memory model, leading to ambiguity about their intended synchronisation behaviour. I developed a fix to the model, and members of ECMA TC39 (in particular Lars T Hansen and Shu-yu Guo) investigated the synchronisation behaviours of current implementations to build confidence in the fix, which was adopted [127]. Unlike two other fixes below, I did not attempt formal verification of this model change, as Atomics.wait and Atomics.notify are implemented using OS primitives which have not yet been formally modelled.

Armv8-A compilation scheme unsoundness This error, the failure of an intended compilation scheme to Armv8-A which is used in practice, was discovered during conversations between myself and Stephen Dolan. I developed a fix to the JavaScript model which I proved correct in Coq with respect to a mixed-size Armv8-A model developed by Christopher Pulte. The fix was adopted into the JavaScript standard [128].

SC condition violation The JavaScript specification explicitly states a correctness condition for the model, that programs with enough synchronisation should provide sequentially consistent (SC) semantics. I discovered that the model violated its stated correctness condition and developed a fix, which was adopted into the standard alongside the above Armv8-A fix [128]. I worked with Guillaume Barbier to verify the fix in Coq. The designers of the model had intended their SC condition to be a realisation of the well-known "Sequential Consistency for Data-Race-Free programs" (SC-DRF) condition [129, 130], which many concurrent languages intend to provide [131, 132, 133]. However, as a consequence of the model's *thin-air* behaviour detailed below, JavaScript's SC guarantees are weaker than true SC-DRF.

Thin-air I discovered that the JavaScript memory model admits undesirable *thin-air executions* [134]. Hans Boehm and Ori Lahav independently observed that JavaScript's thin-air behaviour causes its stated SC condition to be weaker than the desired SC-DRF condition. I then made the closely related observation that this implies that compilation from C++11 to JavaScript, an explicit goal of the JavaScript model's design, is unsound for any reasonable compilation scheme. This deficiency of the model is closely related to the thin-air problem in the C++11 memory model [135]; this issue is known to be impossible to correct in current-generation models [134]. There is ongoing research in the field regarding new iterations of relaxed memory models which avoid the thin-air problem [136, 137, 138].

All of these issues were violations of previously stated explicit design goals for the JavaScript model [129]. Our work developed corrections to the specification for all but the last issue. These corrections have been adopted by the official JavaScript specification. As part of this process, we developed mechanised proofs that the revised model supports the

intended Armv8-A compilation scheme and stated SC property. The final issue cannot be easily corrected, given the current JavaScript relaxed memory model [134]. This is discussed in more detail in §5.7. For now, we make sure any WebAssembly extension of the model does not make the problem worse, and ensure that the model as a whole tracks the best practices of the field so we can benefit from any future developments in the state-of-the-art.

WebAssembly extensions All JavaScript shared memory is created with a fixed size that cannot change. In contrast, WebAssembly memory may be grown dynamically with the **memory.grow** instruction. The informal design of WebAssembly threads intended for memory growth to have sequentially consistent behaviour [139]. However, as I discuss in §5.8, in real implementations, information about the current memory bound may propagate between different WebAssembly threads in a way which is not sequentially consistent. The WebAssembly model must describe this new dimension of relaxed behaviour. Bearing in mind the above formal deficiencies, any extension of the JavaScript model can only be best-effort, and carried out in the knowledge that the JavaScript model must one day be modified to correct the thin-air problem.

5.2 Background

5.2.1 Concurrency on the Web

As discussed earlier, Web content continues to become more computationally demanding. Concurrency has been a recurring proposal over the years to enhance browser-based client-side computational capabilities. In JavaScript, basic support for shared-memory concurrency has already been added, through the SharedArrayBuffer feature. SharedArrayBuffers are low-level buffers of bytes, which can be accessed concurrently in an array-like fashion by "Web workers" (JavaScript's version of threads). Other JavaScript objects may only be accessed by a single thread. When SharedArrayBuffers were first specified, WebAssembly was still under development, and contained no concurrency features. Still, WebAssembly's memory was designed with a future concurrency extension in mind. The WebAssembly threads proposal aims to allow WebAssembly memories to be shared/accessed concurrently. When interoperating with JavaScript, a shared WebAssembly memory will be accessed as a SharedArrayBuffer. Therefore, WebAssembly's concurrency semantics is intrinsically linked to JavaScript's existing concurrency semantics.

5.2.2 Concurrent JavaScript

Fig. 5.1 shows a simple concurrent program, often called the Message Passing (MP) example, expressed as JavaScript, WebAssembly, and Armv8-A. Before discussing the

concurrent behaviour of this example, we will use it to explain how JavaScript declares and accesses concurrent objects. JavaScript's object model has a reputation for complexity. However, SharedArrayBuffers are deliberately very simple objects: they represent a raw zero-initialised memory allocation and provide no innate operations to manipulate the allocated memory. To access a SharedArrayBuffer, it must be wrapped within special "view" object. In the example of Fig. 5.1, the SharedArrayBuffer is created with a size of 8 bytes, and is then wrapped within a *typed array* (in this case an Int32Array). This allows the SharedArrayBuffer to be accessed in an array-like fashion, with the type of the typed array determining the width at which the SharedArrayBuffer is accessed. The 32-bit typed array wrapper means that each array index corresponds to 4 bytes of the underlying SharedArrayBuffer. For example, the access b[0] corresponds to bytes 0-3, while the access b[1] corresponds to bytes 4-7. Note that it is possible for a SharedArrayBuffer to be wrapped by multiple different widths of typed array simultaneously. This can give rise to *mixed-size* accesses which partially overlap with each other.

b = new Int32Array(new SharedArrayBuffer(8));

Thread 1	**Thread 2**
b[0] = 42	L0 = b[1];
b[1] = 42;	L1 = b[0];

<instantiation>

Thread 1	**Thread 2**
(**i32.const** 0) (**i32.const** 42) (**i32.store**)	(**i32.const** 4) (**i32.load**) (**local.set** 0)
(**i32.const** 4) (**i32.const** 42) (**i32.store**)	(**i32.const** 0) (**i32.load**) (**local.set** 1)

---

buffer address in X2

Thread 1	**Thread 2**
MOV W3, #1	
STR W3,[X2, #0]	LDR W0,[X2, #4]
STR W3,[X2, #4]	LDR W1,[X2, #0]

Figure 5.1: A simple concurrent program, expressed in JS, Wasm, and Armv8-a assembly, known as the Message Passing (MP) example.

5.2.3 Relaxed memory

In many real languages and on a variety of hardware, if a program with multiple threads execute loads and stores to memory concurrently, it is common to observe results which cannot be explained by a sequential interleaving of the program's operations. This is known as "relaxed" or "weak" behaviour. As an example, the program of Fig. 5.1 consists

of two threads. As mentioned above, the Int32Array-wrapped SharedArrayBuffer b is zero-initialised on creation. Thread 0 writes 42 to b[0] and b[1] in that order. Thread 1 first reads b[1], then b[0]. If thread 1's read of b[1] assigns the value 1 to local variable L0, in a naive sequentially-interleaved semantics, that must mean that both operations of thread 0 must have already executed. Therefore, thread 1's subsequent read of b[0] would be guaranteed to also set L1 to 1. However, executing this pattern of accesses in (for example) C will commonly result in the outcome L0 = 1, L1 = 0. This could occur for several reasons. First, a compiler could decide to swap the two operations in either thread, which would be a valid single-thread optimisation. Secondly, even if this is forbidden, the underlying behaviour of the hardware can cause this outcome. This could, for example, be the result of thread-local caching of writes, or due to speculative execution in the instruction pipeline causing either thread's operations to take effect out-of-order. When executing the Armv8-A assembly program of Fig. 5.1, this execution is concretely observable. Since the JS and Wasm programs are expected to generate similar Armv8-A code upon compilation, their semantics must allow this outcome. In order to do this, the language must be augmented with a *relaxed memory model*, which describes the space of outcomes which are allowed for a given concurrent program.

Relaxed memory models have been a subject of intense research in recent years [134, 140, 136, 130, 132, 133, 141, 142]. As discussed, a language's relaxed memory model must capture all relaxed outcomes that can result from a combination of compilation optimisations and relaxed behaviours in an underlying architecture. Correctly defining such a model has proven to be a significant challenge. The C++11 relaxed memory model [141, 131] was a seminal work which heavily influenced many subsequent models, including JavaScript.

Like C++11, JavaScript provides both non-atomic operations (the regular accesses of Fig. 5.1), and *atomic operations*, which offer stronger ordering guarantees than regular accesses. While C++11 provides a number of different kind of atomics with varying characteristics, JavaScript only provides the strongest kind: *sequentially consistent*. When the example of Fig. 5.1 has its memory accesses replaced with their sequentially consistent atomic variants (see Fig 5.3), the relaxed outcome L0 = 1, L1 = 0 is no longer permitted. The modern JavaScript specification is admirably precise in its description of the language[1], and the specification of SharedArrayBuffer was accompanied by a official formal relaxed memory model based on a fragment of that of C++11. As hinted by Fig. 5.1 and 5.3, many of the operations of the WebAssembly concurrency proposal mimic those of JavaScript. Therefore, to understand how WebAssembly's concurrency is specified, we must first understand the JavaScript relaxed memory model.

[1] JavaScript's bad semantic reputation is mainly due to legacy behaviours which must be supported for backwards compatibility reasons.

Instructions			
JavaScript	**WebAssembly**	**Armv8-a**	**x86**
_ = b[k]	t.**load**	`ldr`	`MOV`
b[k] = _	t.**store**	`str`	`MOV`
Atomics.load	t.**atomic.load**	`ldar`	`MOV`
Atomics.store	t.**atomic.store**	`stlr`	`XCHG`
Atomics.exchange	t.**atomic.exchange**	`...ldaxr/stlxr...`	`XCHG`

Figure 5.2: Example compilation schemes for JavaScript, and equivalent WebAssembly operations. Arm offers special load-acquire and store-release instructions for the implementation of atomics.

---

b = new Int32Array(new SharedArrayBuffer(8));

Thread 1	**Thread 2**
b[0] = 1	L0 = Atomics.load(b, 1);
Atomics.store(b, 1, 1);	L1 = b[0];

<instantiation>

Thread 1	**Thread 2**
(**i32.const** 0) (**i32.const** 1) (**i32.store**)	(**i32.const** 4) (**i32.atomic.load**) (**local.set** 0)
(**i32.const** 4) (**i32.const** 1) (**i32.atomic.store**)	(**i32.const** 0) (**i32.load**) (**local.set** 1)

---

buffer address in X2

Thread 1	**Thread 2**
MOV W3, #1	
STR W3,[X2, #0]	LDAR W0,[X2, #4]
STLR W3,[X2, #4]	LDR W1,[X2, #0]

Figure 5.3: The example of Fig. 5.1 using atomics. The relaxed outcome is now forbidden.

5.2.4 JavaScript's relaxed memory model

JavaScript's relaxed memory model was designed with several high-level goals in mind [129].

- JavaScript's model must support *mixed-size*, partially overlapping accesses.
- Because of JavaScript's security model (untrusted execution in a user's browser), it cannot treat data races as unconstrained undefined behaviour, as is done in C/C++. The JavaScript model must give defined behaviour to data races, although this behaviour may be extremely weak.
- JavaScript's atomic operations must be compilable according to the existing compilation schemes for C++11 Sequentially Consistent atomics (see Fig. 5.2). Moreover, to support compilation from C/C++ to the asm.js subset of JavaScript, a compilation scheme mapping C++11 atomics to JavaScript atomics and C++11 non-atomics to JavaScript non-atomics must be sound.
- Programs free of data races must have sequentially consistent behaviour. The definition of "data race" ultimately used by the JavaScript model is discussed in §5.3.3.

As mentioned, JavaScript's relaxed memory model is based on a fragment of the C++11 containing non-atomics and sequentially consistent atomics [131, 13]. Differences between the models will be highlighted as appropriate. The style of model used by JavaScript and C++11 is known as an *axiomatic memory model*. With this approach, the semantics of the language is split into two layers: first, a space of *pre-executions* for the given program is defined using an operational semantics, and the pre-executions are then filtered by the axiomatic semantics of the relaxed memory model to obtain a space of *consistent* pre-executions which define the program's observable behaviour. We first describe the *pre-execution* layer. The behaviour of most operations is specified in a familiar operational manner. However, when executing a concurrent load, instead of determining the value returned by the load from concrete state, the semantics picks an arbitrary value to continue execution with, and emits a *read* event recording the load operation together with the choice of value. Similarly, when executing a concurrent store, the semantics emits a *write* event recording the stored value rather than directly mutating the memory state. A single pre-execution consists of the events emitted throughout single execution across all threads, together with basic ordering information derived from the thread-local semantics. The axiomatic part of the relaxed memory model will later determine which choices of loaded values should be permitted, by considering the pre-executions resulting from every possible choice of read value.

5.2.4.1 Pre-executions, formally

Pre-executions are defined formally in Fig. 5.4. The three ordering *modes* correspond to the three strengths of access which are possible in JavaScript. The un ordering corresponds to regular accesses, the sc ordering corresponds to the more strictly ordered sequentially consistent *atomic* accesses, while the init ordering is a special ordering used only by the write which zero-initialises a SharedArrayBuffer. The components of an event, representing a single memory access, are as follows:

- *ord* is the event's ordering mode
- *block* is a symbol used to track which SharedArrayBuffer the event occurred on (a program may create more than one buffer)
- *index* is the byte offset at which the SharedArrayBuffer was accessed
- *reads* is the list of bytes observed by the event
- *writes* is the list of bytes written by the event – some events (such as the one generated by `Atomics.exchange`) may perform both a read and a write
- *tearfree* is a special parameter which fixes whether an event may appear to split into parts – for example a 64-bit access implemented on a platform with only 32-bit width native accesses may need to be implemented as a pair of accesses, which may be observed independently

We assign unique ids to events to ensure they are distinct. In the text of the JavaScript specification, it is assumed that events created by different execution steps are a-priori distinct from each other even if all of their fields are identical, so this field is not explicitly included.

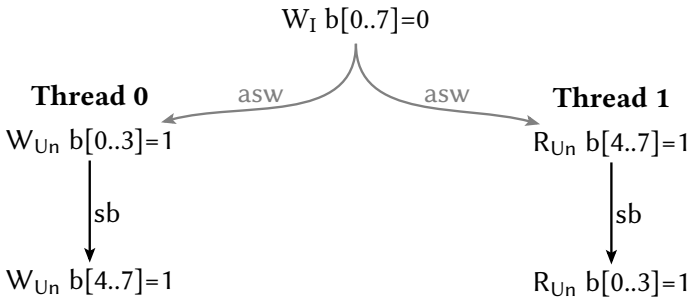
$$
\begin{array}{rcl}
\mathit{mode} & ::= & \mathsf{Unordered} \;//\; \mathsf{un} \mid \mathsf{SeqCst} \;//\; \mathsf{sc} \mid \mathsf{Init} \;//\; \mathsf{init} \\
\mathit{addr}, \mathit{id} & ::= & \text{an infinite set of abstract names} \\
\mathit{event} & ::= & \{\ \text{ord} :: \mathit{mode},\ \text{reads} :: \mathit{list\ byte}, \\
 & & \quad \text{block} :: \mathit{addr},\ \text{writes} :: \mathit{list\ byte}, \\
 & & \quad \text{index} :: \mathit{nat},\ \text{tearfree} :: \mathit{bool}\ \}_{\mathit{id}} \\
\mathit{exec} & ::= & \{\ \text{evs} :: \mathit{set\ event}, \\
 & & \quad \text{sequenced-before} \;//\; \mathsf{sb} :: \mathit{set}\ (\mathit{event} \times \mathit{event}), \\
 & & \quad \text{additional-synchronizes-with} \;//\; \mathsf{asw} :: \mathit{set}\ (\mathit{event} \times \mathit{event})\ \}
\end{array}
$$

Figure 5.4: JavaScript Pre-execution. Abbreviations are marked by "//".

b = new Int32Array(new SharedArrayBuffer(8));

Thread 0	**Thread 1**
b[0] = 1	L0 = b[1];
b[1] = 1;	L1 = b[0];

Outcome: L0 = 1, L1 = 1



Outcome: L0 = 1, L1 = 0

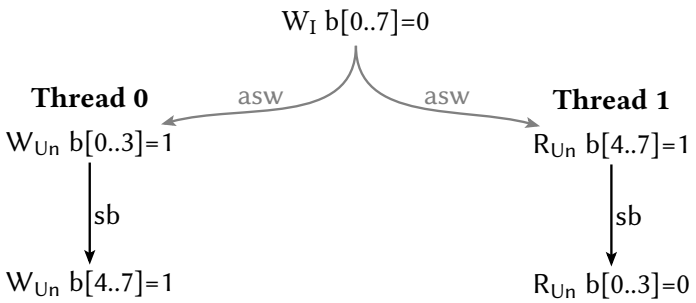


Figure 5.5: A simple JS concurrent program (MP), together with two pre-executions.

Aside from the set of events, a pre-execution consists of the *sequenced-before* relation, which tracks intra-thread execution order, and *additional-synchronized-with*, which represents strong ordering constraints imposed by synchronous inter-thread communication such as using the `postMessage` operation. Both of these are illustrated by example in Fig. 5.5, which shows the JS program of Fig. 5.1 together with two pre-executions. The pre-executions are depicted as graphs, with the events of each execution represented as nodes. For example, the node Wun b[0-3]=1 is an Unordered event writing 1 as a 4 byte value, starting at index 0, to the SharedArrayBuffer `b`. Note the sequenced-before edges between consecutive events in the same thread. When the SharedArrayBuffer is transmitted to the two worker threads, the actions of these threads take place "after" the transfer. The additional-synchronizes-with edges represent this ordering.

The second layer of semantics is the relaxed memory model, a series of axiomatic constraints over the pre-executions. The constraints of the axiomatic memory model are formally defined over a *candidate execution*, which is a combination of a pre-execution with an existentially quantified *execution witness*. Intuitively, the execution witness is a collection of relations between events of the pre-execution, which explains why each concurrent read is allowed to take the value that it did. The formal structures of the execution witness and candidate execution are given in Fig. 5.6.

The core of the execution witness is a *reads-byte-from* relation. Each byte of a read event is linked to a write event on the same location, such that the written value matches the read value. The permitted shapes of the reads-byte-from relation depend on various ordering constraints that must be described by the memory model. In addition, the execution witness must fix a total order (*total-order*) over all events, which is used to constrain the possible behaviours of the stronger SC atomic operations. The reads-byte-from relation is similar to the *reads-from* relation of C++11. The key difference is that the C++11 relaxed memory model does not support mixed-size behaviours, and so its reads and writes are simply paired according to matching abstract locations. In JavaScript, each individual byte of the read and write must be separately related, since a single read may observe individual bytes from multiple writes. JavaScript defines its own version of reads-from, derived from reads-byte-from by projecting away the byte component (Fig. 5.6). Note that by convention the reads-from relation relates writes (as the left component) to reads (as the right component). The model also defines two other derived relations. The *synchronizes-with* relation represents the strong ordering guarantees given by an atomic read. If an atomic read gets its value from an atomic write of equal range, or from an initializing write, then a synchronizes-with edge is created to represent that there is an ordering constraint between prior events in the writing thread, and subsequent events in the reading thread. The *happens-before* relation is the main definition used to state the constraints of the model. It is the transitive closure of the intra-thread program order, the synchronizes-with relation, combined with the requirement that all accesses

must be ordered after any overlapping initial writes.

It is specified that an implementation may only exhibit behaviours for which there exists a corresponding *well-formed* candidate execution in the semantics satisfying the models' *consistency* predicates. Well-formedness of a candidate execution enforces the basic characteristics of the core witness relations – total-order must be a strict total order, and reads-byte-from may only relate a write with a read if they access the same location. A formal definition of well-formedness is given in Fig. 5.7.

$$\begin{array}{rcl} \mathit{witness} & ::= & \{\ \text{reads-byte-from} \mathbin{/\!/} \mathbf{rbf} :: \mathit{set}\ (\mathit{nat} \times \mathit{event} \times \mathit{event}), \\ & & \ \ \text{total-order} \mathbin{/\!/} \mathbf{tot} \quad :: \mathit{set}\ (\mathit{event} \times \mathit{event})\ \} \end{array}$$

$$\mathit{candidate\_exec} \triangleq \mathit{exec} +\!\!+ \mathit{witness}$$

$$\begin{array}{rcl} \text{range}_r(E : \mathit{event}) & \triangleq & [\ E.\text{index} \ldots E.\text{index} + |E.\text{reads}|\ ) \\ \text{range}_w(E : \mathit{event}) & \triangleq & [\ E.\text{index} \ldots E.\text{index} + |E.\text{writes}|\ ) \\ \text{range}(E : \mathit{event}) & \triangleq & \text{range}_r(E) \cup \text{range}_w(E) \\ \text{write}(E : \mathit{event}) & \triangleq & (E.\text{writes} \neq []) \\ \text{overlap}(E_1, E_2 : \mathit{event}) & \triangleq & E_1.\text{block} = E_2.\text{block} \wedge \text{range}(E_1) \cap \text{range}(E_2) \neq \varnothing \end{array}$$

Derived relations *(w.r.t. a candidate execution)*

$$\text{reads-from} \mathbin{/\!/} \mathbf{rf} \triangleq \{\langle A, B\rangle \mid \exists k. \langle k, A, B\rangle \in \text{reads-byte-from}\}$$

$$\text{synchronizes-with} \mathbin{/\!/} \mathbf{sw} \triangleq \left\{ \langle A, B\rangle \;\middle|\; \begin{array}{l} \langle A, B\rangle \in \text{reads-from} \wedge B.\text{ord} = \mathsf{SeqCst} \wedge \\ \left(\begin{array}{l} (\text{range}_w(A) = \text{range}_r(B) \ \wedge A.\text{ord} = \mathsf{SeqCst}) \vee \\ (\forall C.\ \langle C, B\rangle \in \text{reads-from} \implies C.\text{ord} = \mathsf{Init}) \end{array} \right) \end{array} \right\} \cup \text{additional-synchronizes-with}$$

$$\text{happens-before} \mathbin{/\!/} \mathbf{hb} \triangleq \left(\begin{array}{l} \text{sequenced-before} \cup \text{synchronizes-with} \cup \\ \{\ \langle A, B\rangle \mid A.\text{ord} = \mathsf{Init} \wedge \text{overlap}(A, B)\ \} \end{array} \right)^+$$

Figure 5.6: JavaScript candidate execution definition and derived relations. We introduce short names for some relations after the "//".

$$\text{well-formed}(k : \mathit{nat}, E_w : \mathit{event}, E_r : \mathit{event}) \triangleq$$
$$k \in \text{range}_w(E_w) \wedge k \in \text{range}_r(E_r) \wedge E_w.\text{block} = E_r.\text{block} \wedge$$
$$(E_w.\text{writes})[k - E_w.\text{index}] = (E_r.\text{reads})[k - E_r.\text{index}] \wedge E_w \neq E_r$$

$$\text{well-formed}(\mathit{rbf} : \mathit{set}\ (\mathit{nat} \times \mathit{event} \times \mathit{event}), \mathit{Es} : \mathit{set}\ \mathit{event}) \triangleq$$
$$(\ \forall E_r \in \mathit{Es}, k \in \text{range}_r(E_r).\ \exists! E_w.\ \langle k, E_w, E_r\rangle \in \mathit{rbf}\ ) \wedge$$
$$(\ \forall E_r \in \mathit{Es}, k \notin \text{range}_r(E_r).\ \nexists E_w.\ \langle k, E_w, E_r\rangle \in \mathit{rbf}\ ) \wedge$$
$$\forall \langle k, E_w, E_r\rangle \in \mathit{rbf}.\ \text{well-formed}(k, E_w, E_r)$$

$$\text{well-formed}(\mathit{CE} : \mathit{candidate\_exec}) \triangleq$$
$$\mathit{CE}.\text{total-order is a strict total order on } \mathit{CE}.\text{evs} \wedge$$
$$\text{well-formed}(\mathit{CE}.\text{reads-byte-from}, \mathit{CE}.\text{evs})$$

Figure 5.7: Candidate execution well-formedness

Happens-Before Consistency (1):
$$\text{happens-before} \subseteq \text{total-order}$$

Happens-Before Consistency (2):
$$\forall E_w E_r.\ \langle E_w, E_r\rangle \in \text{reads-from} \implies \neg(E_r\ \text{happens-before}\ E_w)$$

Happens-Before Consistency (3):
$$\forall \langle k, E_w, E_r\rangle \in \text{reads-byte-from}.$$
$$\nexists E'_w.\ (E_w\ \text{happens-before}\ E'_w) \wedge$$
$$(E'_w\ \text{happens-before}\ E_r) \wedge k \in \text{range}_w(E'_w)$$

Tear-Free Reads:
$$\forall E_r.\ E_r.\text{tearfree} \implies$$
$$\left|\left\{\ E_w\ \middle|\ \begin{array}{l}\langle E_w, E_r\rangle \in \text{reads-from} \ \wedge\ E_w.\text{tearfree} \wedge \\ \text{range}_w(E_w) = \text{range}_r(E_r)\end{array}\right\}\right| \leqslant 1$$

Sequentially Consistent Atomics (first attempt):
$$\forall E_w E_r.\ E_w\ \text{synchronizes-with}\ E_r \implies$$
$$\nexists E'_w.\ (E_w\ \text{total-order}\ E'_w) \wedge (E'_w\ \text{total-order}\ E_r) \wedge \text{range}_w(E'_w) = \text{range}_r(E_r)$$

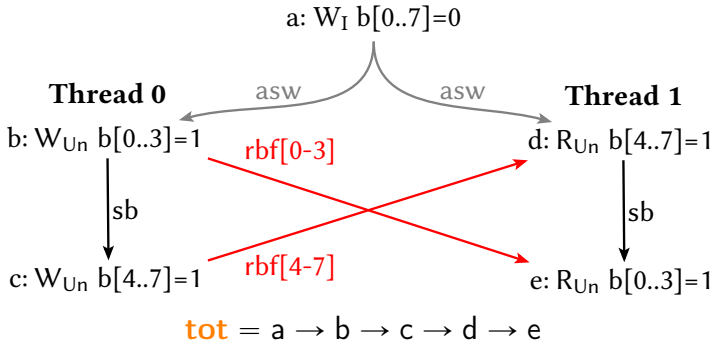
Figure 5.8: Candidate execution consistency as originally defined by the JavaScript specification.

The axiomatic constraints defining consistency of a given candidate execution are given in Fig. 5.8. First, the happens-before order must be consistent with the total-order. Second, it is not permitted for a read to get its value from a write if the read is ordered before the write according to happens-before. Finally, it is not permitted to read from a "stale" write, where a newer write to the same range exists according to happens-before. The **Tear-Free Reads** condition ensures that a read which has been marked tearfree will not combine the individual bytes of same-range tearfree writes. The **Sequentially Consistent Atomics** condition further constrains the permitted orderings of atomic accesses. The informal intention is that atomic accesses should appear sequentially consistent unless they race with a non-atomic access, or an atomic access with a different range. This latter relaxation allows 64-bit atomic accesses to be implemented with locks on 32-bit machines without requiring smaller possibly overlapping atomic accesses to take the same locks. However, as we will discuss, the condition as stated is flawed in several ways and must be corrected.

b = new Int32Array(new SharedArrayBuffer(8));

Thread 0	**Thread 1**
b[0] = 1	L0 = b[1];
b[1] = 1;	L1 = b[0];

Outcome: L0 = 1, L1 = 1



Outcome: L0 = 1, L1 = 0

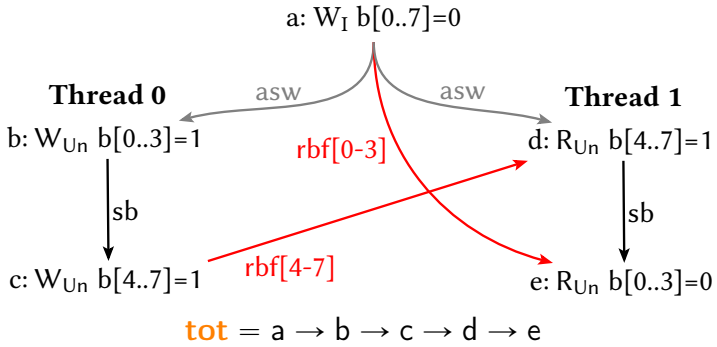


Figure 5.9: A simple JS concurrent program (MP), together with two consistent candidate executions.

As an illustrative example, the pre-executions of Fig. 5.5 are extended to candidate executions in Fig. 5.9. Both of these candidate executions are consistent according to the model. As discussed earlier, the second execution demonstrates a relaxed behaviour, which must be allowed as it is concretely observable when compiling to Arm. Going forward, when depicting candidate executions graphically in this way, edges not relevant to the example under discussion will be elided.

5.2.4.2 JS vs C++11

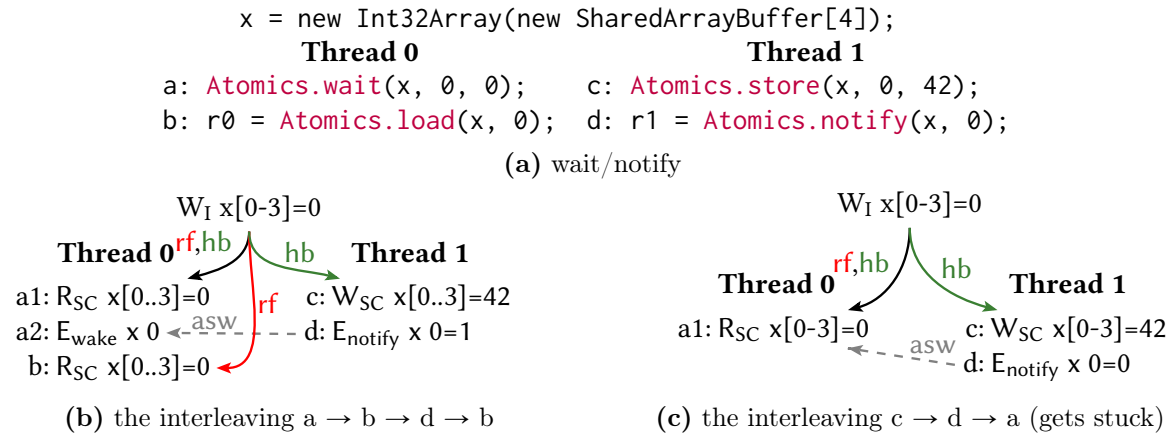
As discussed, many of JavaScript's definitions are based on those of C++11. JavaScript's key generalisation is that its fundamental witnessing relation, *reads-byte-from*, is bytewise instead of relating reads and writes one-to-one. Because C++11 accesses can never partially overlap, its SC atomics always synchronise if they are related by *reads-from*. In contrast, JavaScript's atomics only synchronise if their ranges are identical.

Aside from this, JavaScript's model is mainly notable for what it leaves out from C++11, with subtle consequences. Unlike C++11, there is no stipulation that programs with data races exhibit undefined behaviour; such programs in JavaScript simply exhibit the behaviours allowed by the core model. As mentioned, this was a deliberate decision made with the intent of ensuring that a malicious website could not compromise a visitor by tricking them into executing JavaScript code containing a data race. This means that JavaScript's model must be sure to correctly describe many behaviours only observable in racy programs which are not considered to have defined behaviour in C++11. In particular, an issue with the JavaScript model's intended Armv8-A compilation scheme (§5.3.2) is the result of a racy behaviour which was not correctly modelled. Moreover, JavaScript does not include a C++11 constraint on non-atomics known as *Last Visible Side-Effect*, since this condition is only correct if data races are undefined behaviour. This leads to an incompatibility with the C++11 model which we describe in §5.7.

We now discuss the issues found in the JavaScript model, and, where possible, their associated corrections.

5.3 Correcting the JavaScript memory model

In the course of attempting to extend JavaScript's memory model to WebAssembly, we discovered several deficiencies in the model. Wherever possible, we corrected these errors, and helped get them adopted as changes to JavaScript's official specification. First, we will discuss errors found in JavaScript's model that were corrected, and the subsequent verification which took place. Then, we will conclude by describing an outstanding issue in the model which is related to the notorious *out-of-thin-air* problem [134].



(a) wait/notify

(b) the interleaving a → b → d → b

(c) the interleaving c → d → a (gets stuck)

Figure 5.10: These two candidate executions for are forbidden if the model adds the grey edges.

5.3.1 Omission of wait/notify synchronisation

In addition to the memory operations previously described, JavaScript defines the thread synchronization operations `Atomics.wait` and `Atomics.notify`. We explain these operations by way of an example program (Fig. 5.10a). All operations are to the same location on the SharedArrayBuffer in `x`. The `Atomics.wait` operation reads memory location 0, and compares the result to an expected value, 0. If the expected value does not match the read value, execution continues as normal. If the expected value matches the read value, the thread suspends execution, placing itself in a *wait queue* associated with the read location. The `Atomics.notify` operation of Thread 1, to the same location, will wake all threads in the wait queue for that location. The return value of `Atomics.notify` is the number of threads woken.

Intuitively, this program should always terminate, with the load of line (b) guaranteed to read 42. If Thread 0 executes `Atomics.wait` first, it will suspend until both (c) and (d) have executed, meaning (b) will not execute until 42 has been written. Alternatively, if Thread 1 executes `Atomics.notify` first, then it will already have executed line (c) and written 42 in location 0. Therefore Thread 0's `Atomics.wait` should continue execution as it does not observe its expected value. However, this intuition relies on the operations providing ordering guarantees, through synchronization, which are not explicitly represented in the axiomatic memory model.

Interactions with the wait queue are specified purely using an interleaving of the thread-local semantics. The specification informally describes threads updating the wait queue as entering and leaving a lock-like "critical section". However, it does not describe how the interleaved order of critical section entries affects the candidate executions permitted by the axiomatic memory model. We correct this so that entering the critical section creates synchronization edges in the candidate execution to all previous exits. This is in line with the treatment of locks in C/C++11 and the *monitor lock* of the Java memory model.

It also fits the informal understanding of JavaScript implementers, who reported that they currently implement lock-like synchronization for `Atomics.wait/notify` [143].

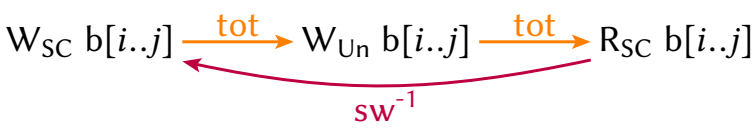
These additional synchronization edges are necessary to ensure that the axiomatic model correctly forbids intuitively disallowed executions. Fig. 5.10b shows an undesirable execution where (b) reads 0 even though it cannot have executed until (d) notifies (a). Similarly, in Fig. 5.10c, (a) reads 0, suspending, even if (d) records that there were no threads notified, meaning (c) must have already executed. We modify JavaScript's construction of pre-executions to ensure that the the critical section entry ordering guarantees are explicitly represented as additional-synchronizes-with edges (given by the dashed grey lines), ensuring that these executions are forbidden.

5.3.2 Armv8-A compilation

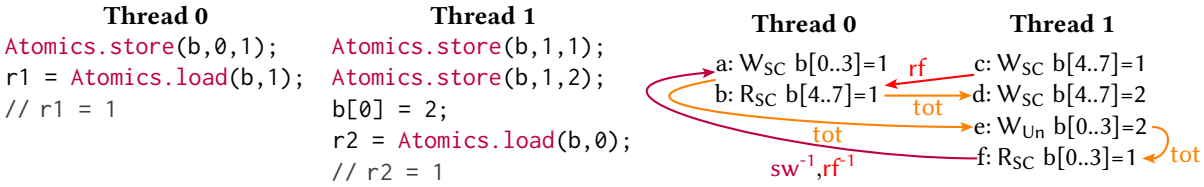
SharedArrayBuffer accesses in a JavaScript program are ultimately compiled to the native assembly accesses of the user's architecture. These assembly languages have their own relaxed memory models, and it is important that the compilation scheme converting JavaScript accesses to plaform assembly accesses respects the semantics of JavaScript's relaxed memory model. That is, it should not be possible to write a JavaScript program which, when compiled to platform assembly, is permitted by the assembly language's memory model to exhibit a behaviour which the JavaScript model forbids.

The intended compilation schemes from JavaScript accesses to x86 and Armv8-A accesses are given in Fig. 5.2. Note that these schemes do not take into account any compiler optimisations which may also be applied. It was an explicit goal of the memory model's original design that these compilation schemes should be supported, as they are already supported by C++11. Stephen Dolan and I discovered that the JavaScript model as originally specified exhibits a clear counter-example to the Armv8-A compilation scheme. A refined version of the counterexample, found automatically through model-checking (see §5.5) is shown in Fig. 5.11. The outcome `r1 = 1, r2 = 1` is forbidden by the JavaScript memory model, but is concretely observable in Armv8-A given the desired compilation scheme.

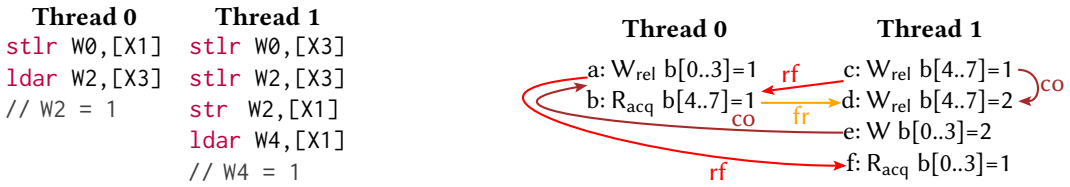
The core issue with the JavaScript model that enables this counter-example is the **SC atomics** condition. While intuitively it was intended to restrict the behaviour of SC atomics, it also ends up over-constraining the behaviour of non-atomics. Consider the following shape, which is forbidden by the **SC atomics** rule of Fig. 5.8.



In particular, note that the shape is forbidden even though the middle W is non-atomic.



(a) an outcome forbidden by JavaScript



(a) when the program is compiled to Armv8-A, the outcome is allowed

Figure 5.11: A JavaScript program which violates the memory model when compiled to Armv8-A.

This shape occurs between events a, e, and f of the JavaScript execution shown in Fig. 5.11a, and therefore the execution is forbidden. Note that no other candidate execution can make this output observable, since alternative configurations of edges are also forbidden by the memory model. In particular, because the event (b) reads 1, there must be a total-order edge from (b) to the write (d). If the edge were the other way around, (b) would not be allowed to read 1, and could only read 2 from (d), since reading from (c) would be forbidden by the **Sequentially Consistent Atomics** rule. Therefore the total-order edge from (a) to (e) is also fixed, because of total-order's transitivity and the fact that happens-before is a subset of total-order.

Without getting into the details of the Armv8-A memory model, the accesses corresponding to events (d) and (e) are allowed by Armv8-A to occur out-of-order. If these events are swapped in program order, then the execution is allowed as a simple sequential interleaving of the accesses.

The fix is simple: the condition must be weakened to only apply when the middle write is atomic. This leads to the following revised condition:

Sequentially Consistent Atomics (first revision):

$$\forall E_w E_r.\ E_w \text{ synchronizes-with } E_r \implies$$
$$\nexists E'_w.\ E'_w.\text{ord} = \mathsf{sc} \wedge (E_w \text{ total-order } E'_w) \wedge (E'_w \text{ total-order } E_r) \wedge$$
$$\text{range}_w(E'_w) = \text{range}_r(E_r)$$

It is possible this issue arose by uncritically importing a similar condition from a draft version of the C++11 model [131]. In C++11, non-atomic writes are more restricted in where they can occur: an analogous program to our Armv8-A counter-example in C++11 would have undefined behaviour (due to the presence of a data race) and therefore the axiomatic rules do not need to explicitly forbid this shape.

5.3.3 SC-DRF

Sequential Consistency for Data-Race-Free programs (SC-DRF) is a well-known correctness condition for relaxed memory models, which is designed to allow programmers to reason about a restricted subset of programs as if the memory model contains no relaxed behaviours. It states that if a program has no "data races", then it will only exhibit behaviours which are consistent with a naïve sequential interleaving of its operations (a "sequentially consistent execution"). In the classic statement of the property by Adve and Hill [130], a program is considered to have a data race if it admits a sequentially consistent execution such that there are two accesses to the same location in different threads, at least one of which is a write, without some explicit synchronization or strong ordering between them (as embodied by a relation such as *happens-before*). If a program can be determined data-race-free, a programmer can reason about its behaviour using the naïve sequential interleaving semantics, without needing to understand the full underlying relaxed memory model of the language. SC-DRF is considered by many to be *the* desirable correctness condition for a relaxed memory model, and has received considerable research attention [144, 130, 132, 145].

5.3.4 SC-DRF in C++11

C++11 specifies its own version of "data race", and declares any program exhibiting one to have undefined behaviour. Its definition of data race is based on finding a race in the consistent (according to the relaxed memory model) candidate executions of the program. A program is considered to exhibit a data race if it admits a valid candidate execution which contains two memory model events accessing the same location, at least one of which is a write, and at least one of which is non-atomic, with no *happens-before* relation between them. Using this definition undercuts much of the SC-DRF property's usefulness for abstracting away the full memory model, since a program cannot be determined data-race-free in the C++11 sense without reasoning about its consistent executions, according the memory model itself.

The circumstances under which a "C++11-style" definition of SC-DRF coincides with the classic statement are subtle. C++11's data race definition only applies when the race involves a non-atomic, however the language also provides a number of "low-level atomics" which do not have SC behaviours, but do not trigger data-race-based undefined behaviour. Batty et al. [134] show that for a subset of C++11 containing only non-atomics and SC atomics, the two definitions of SC-DRF coincide. For a larger fragment of C++11, its SC-DRF definition is known to be weaker than the classic statement [144], as a deliberate design decision.

5.3.5 SC-DRF in JavaScript

The JavaScript specification explicitly states its own SC-DRF correctness condition as part of its memory model. JavaScript's statement of SC-DRF is closely based on that of C++11. As we will discuss in §5.7, we now know that JavaScript's statement of SC-DRF is weaker than that of C++11. For now, we discuss the definition of JavaScript's stated SC-DRF property, and the discovery and correction of errors in the model which led to a violation of even JavaScript's weaker property.

Two events A and B in a given candidate execution exhibit a **Data Race** iff:

$$\begin{array}{l}(A.\text{ord} = \mathsf{un} \vee B.\text{ord} = \mathsf{un} \vee \text{range}(A) \neq \text{range}(B)) \wedge \\ \quad \text{overlap}(A, B) \wedge (\text{write}(A) \vee \text{write}(B)) \wedge A \neq B \wedge \\ \quad\quad \neg(A \text{ happens-before } B \vee B \text{ happens-before } A)\end{array}$$

Figure 5.12: Definition of a JavaScript data race according to the standard.

5.3.5.1 JavaScript data race

The JavaScript specification formalises the property of a candidate execution having a *data race* as shown in Fig. 5.12. An execution is said to contain a data race if it contains two events where their locations overlap, at least one event is a write, and also either at least one event is a non-atomic access, or both events are atomics which only partially overlap. Most of this definition is identical to the data race condition defined in C++11 memory model. However, there are two main differences.

First, the C++11 model does not define the behaviour of mixed-size accesses. A data race is never allowed to occur between two SC atomic accesses to the same location. JavaScript however explicitly allows data races to occur between atomic accesses if they only partially overlap. This is consistent with the general principle of the JavaScript model that atomics only "act like" atomics in situations which do not involve mixed-size behaviour. For example, matched read-write atomics create a *synchronizes-with* edge only if their footprints precisely match.

Second, in C++11 the presence of a data race immediately means that the entire program has undefined behaviour. In JavaScript, programs with data races still have the defined behaviour ascribed to them by the axiomatic memory model, although this behaviour may be very weak. The SC-DRF condition simply describes the "good behaviour" of programs without these data races.

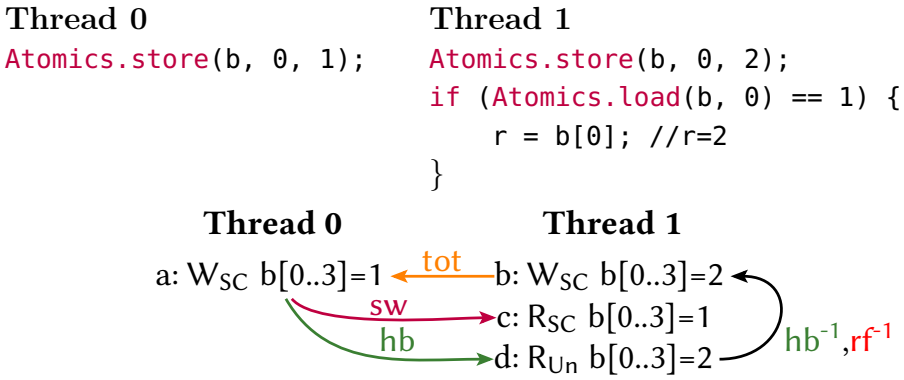
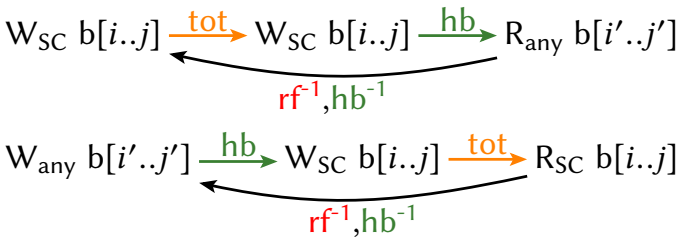


Figure 5.13: SC-DRF violation by a JavaScript program

5.3.5.2 The SC-DRF property

The SC-DRF property as given in the JavaScript specification states that if the program is *data-race-free* (that is, none of its executions contain a data race according to Fig. 5.12), then every consistent execution of the program will be sequentially consistent (that is, explainable as a naive sequential interleaving of the program's accesses). In the JavaScript model as given above, we found a simple counter-example to this property, shown in Fig, 5.13. No sequential interleaving of the program's accesses can explain the model-allowed behaviour where the non-atomic access of Thread 1 reads the value 2. The issue is that the *SC Atomics* consistency condition is not strong enough. The constraint only applies when the read-write pair of events are both atomics. However, an additional constraint is needed where one part of the read-write pair is a non-atomic which is fully synchronized through other means (in this case, an intervening atomic read-write pair). Interestingly, a similar issue was discovered and corrected in an early draft of the C++11 model, caused by mixing non-atomic writes (caused by initialisation) with SC atomic reads on the same location, as documented by Batty et al. [141]. We can fix the JavaScript model in a similar way, by disallowing the following shapes:



Note that the former of these shapes occurs in the spuriously allowed execution of Fig. 5.13.

Combining this with the previous Armv8-A compilation fix, we arrive at the version of the **SC Atomics** condition I proposed (Fig. 5.14), which was adopted by the JavaScript standards committee.

Sequentially Consistent Atomics (final):

$$\forall E_w E_r.\, \langle E_w, E_r \rangle \in \text{reads-from} \ \wedge\ E_w \text{ happens-before } E_r \implies$$
$$\nexists E'_w.\, E'_w.\text{ord} = \mathsf{SeqCst} \ \wedge\ E_w \text{ total-order } E'_w \ \wedge\ E'_w \text{ total-order } E_r \ \wedge$$
$$\left(\begin{array}{l} \quad (\text{range}_w(E'_w) = \text{range}_r(E_r) \ \wedge\ E_w \text{ synchronizes-with } E_r) \\ \vee\ (\text{range}_w(E_w) = \text{range}_w(E'_w) \wedge E_w.\text{ord} = \mathsf{SeqCst} \ \wedge\ E'_w \text{ happens-before } E_r) \\ \vee\ (\text{range}_w(E'_w) = \text{range}_r(E_r) \ \wedge\ E_w \text{ happens-before } E'_w \ \wedge\ \ E_r.\text{ord} = \mathsf{SeqCst}) \end{array} \right)$$

Figure 5.14: Corrected **SC Atomics** rule as adopted into the standard.

5.4 Verifying the corrected model

We have two main verification aims.

- Prove that the revised JS model supports the desired compilation scheme to Armv8-A
- Prove that the revised JS model satisfies the stated SC-DRF correctness condition

In both cases, as our proposed changes to the JavaScript model were iterated on, these properties were checked up to a finite bound using the Alloy model checker, before being given a full proof in Coq once we had confidence in a final version of the proposed changes. This process will be described below.

It is worth clarifying some limiting assumptions which were made during this section, which mirror assumptions made by similar state-of-the-art proofs in existing work. First, all JavaScript candidate executions were assumed to access only a single SharedArrayBuffer, created ahead of time. JavaScript makes strong ordering assumptions about the initial writes performed during creation of a SharedArrayBuffer, and we cannot verify these assumptions without a richer concurrent model of the OS than is available in existing work. By assuming that all accesses are to the same SharedArrayBuffer, we remove the case where a SharedArrayBuffer is initialised concurrently with an access to another SharedArrayBuffer. This is somewhat analogous to the assumptions in many existing works on platform assembly memory models that do not handle executions which alter the virtual memory mapping or raise an exception.

Second, a minor modification is made to the way atomic Read-Modify-Write (RMW) accesses are modelled in JavaScript. Instead of being a single event that both reads and writes, an RMW access is modelled as a pair of separate read and write events, which are forced to occur adjacent to each other in all relevant orderings. A single JavaScript RMW is concretely compiled to a pair of Armv8-A load/store exclusives, and this transformation of the JavaScript model allows the execution events associated with the RMW to be mapped one-to-one with the events of the load/store exclusive pair. This RMW transformation is often used when conducting proofs about the C++11 model for the same reasons [144, 146].

Third, all JavaScript accesses are assumed to be aligned. This is guaranteed by the commonly-used TypedArray API, but a lower-level DataView API exists which permits

unaligned accesses to an underlying SharedArrayBuffer. Therefore these proofs should be understood as concerning the compilation of TypedArray accesses, rather than the more complex DataView accesses. This assumption, combined with the RMW transformation above, simplifies the Armv8-A proof, as regular JavaScript memory access events can be mapped one-to-one with Armv8-A memory access events. In the case that a single JavaScript access is not aligned, the corresponding access in Armv8-A would have to be represented in its model by multiple bytewise events.

5.4.1 Armv8-A mixed-size model

At the time the work was carried out, no official mixed-size axiomatic model for Armv8-A relaxed memory existed. In order to make the compilation scheme correctness proof more tractable, Christopher Pulte developed an mixed-size axiomatic model for Armv8-A. Below, I describe the validation of this model, which I carried out in collaboration with Christopher. The model is given in full in Appendix A.

The two starting points for developing the mixed-size axiomatic model are the existing Flat model [147, 148], an operational model with mixed-size support that is part of the `rmem` tool [149], and Arm's reference model [150, 147, 151], an axiomatic specification defined in herd [152], without mixed-size support. The two models are based on extensive past research on architectural concurrency for Armv8-a (and related Power), discussion with architects, and experimental hardware testing [147, 148, 153, 154, 152, 155, 156, 157, 158, 149, 159, 160, 161, 162, 163]. Our mixed-size axiomatic model generalises the reference axiomatic model to mixed-size programs in a way that aims to follow the Flat model's behaviour – Flat has been developed in collaboration with Arm and is extensively experimentally validated, although it is beyond the scope of this work to further investigate the correctness of Flat itself. In particular, see Alglave et al. [164] for a critique of Flat in the context of more recent work in defining a mixed-size Armv8-A model.

In cases where Flat's mixed-size semantics is still potentially subject to change, our axiomatic Armv8-A model attempts to over-approximate, so that it is possible that our model allows some mixed-size behaviours which are not currently allowed by Flat. As long as our model is *no stronger than* Flat, however, any compilation scheme our Armv8-A model supports will also be supported by the Flat model. Since we aim to use our model primarily to investigate the compilation scheme correctness of JavaScript, this "no stronger than Flat" property is what we focus on experimentally validating, using an extensive corpus of tests.

In Pulte et al. [147], the uni-size axiomatic and Flat operational model were hand-proved equivalent (for uni-size input programs). Formally proving a correspondence between mixed-size Flat and our mixed-size axiomatic model would be a substantial effort in its own right: extending the axiomatic model to mixed-size accesses breaks some

assumptions made by the existing proof. Extending the proof is beyond the scope of this thesis, and further work still needs to be done in order to find axiomatic rules that are precisely equivalent to Flat. However, we believe that our approach of generalising an existing uni-size axiomatic model, combined with extensive validation, represents an important first step in solving this more general problem.

The experimental validation is based on the corpus of litmus tests from prior work on Armv8-A (the majority systematically generated with `diy` [163], and including hand-written tests used in Flur et al. [148, 153]). We run the Flat model on this test suite and enumerate, for each test, the set of all behaviours allowed by Flat. We instrument the Flat model to generate, for each such possible outcome, the candidate execution corresponding to the operational model's trace. We log the candidate executions, and feed them into our Alloy implementation of the Armv8-A axiomatic model (see §5.5) to ensure the *soundness* of the axiomatic model: that it allows each such Flat-allowed execution.

The litmus test suite we rely on contains 11,587 litmus tests. We run the tests on a Ubuntu 18.04.2 POWER9 machine (160 CPUs at 2.9GHz, 125GB ram) with no memory limit and a 168 hour time limit. Of the 11,587 tests, 11,578 successfully run to completion in Flat (2635 mixed-size and 8943 non-mixed-size). For the 9 tests where Flat does not complete, 3 are due to instructions currently unsupported by Flat, 4 running out of memory, 1 running out of time, and a final test crashing with an unspecified error. The 11,578 tests where Flat successfully completes generate a total of 167,014 candidate executions. We run the mixed-size Alloy-based Armv8-A axiomatic model (§5.5) on these and confirm that it allows every such Flat-allowed execution.

5.5 Alloy verification

For the SC-DRF and Armv8-A compilation issues described in §5.3.3 and §5.3.2, we define the JavaScript and mixed-size Armv8-A models in the Alloy model checker [11], allowing us to compare the two models and investigate whether individual litmus tests are allowed by the models. This approach was first used by Wickerson et al. [165]. While they took existing uni-size models, written in herd [152], and automatically converted them to Alloy, we directly transcribe the JavaScript (corrected and uncorrected) and Armv8-A models into Alloy by hand. Alloy's syntax supports arbitrary first-order predicates, so the models can be faithfully reproduced.

5.5.1 Armv8-A search

We are able to use these Alloy models to test that our hand-found counter-examples are real (i.e. that the execution is disallowed in JavaScript but the related execution is allowed in our Armv8-A model). In addition, following the approach of Wickerson et al. [165], we

are able to use Alloy to automatically find smaller counter-examples than we were able to find manually. Our smallest hand-discovered counter-example for the Armv8-A violation required 8 events and 3 byte locations; Alloy finds a counter-example with 6 events, 2 byte locations (Fig. 5.11).

In this search, we are looking for counter-examples to the Armv8-A compilation scheme. Such a counter-example is an execution $Exec_{\text{JS}}$ of a JavaScript program $Prog_{\text{JS}}$ that is invalid according to the JavaScript memory model, but which corresponds to an execution $Exec_{\text{ARM}}$ of a program $Prog_{\text{ARM}}$ obtained by compiling $Prog_{\text{JS}}$ to Armv8-A, and where $Exec_{\text{ARM}}$ is allowed by the Armv8-A concurrency model.

To this end, as is done by Wickerson et al. [165], we define a translation relation on candidate executions. Intuitively this should relate a JavaScript execution $Exec_{\text{JS}}$ with an Arm execution $Exec_{\text{ARM}}$ if $Exec_{\text{JS}}$ and $Exec_{\text{ARM}}$ are executions of the programs $Prog_{\text{JS}}$ and $Prog_{\text{ARM}}$, respectively, such that $Prog_{\text{JS}}$ compiles to $Prog_{\text{ARM}}$, and $Exec_{\text{JS}}$ and $Exec_{\text{ARM}}$ have the same observable behaviour. We define a translation relation, that:

- is compatible with the compilation scheme:
  events in $Exec_{\text{JS}}$ arising from JavaScript accesses are related to events in $Exec_{\text{ARM}}$ arising from the compiled Armv8-A accesses;
- is compatible with the program structure:
  it preserves sequenced-before edges (maps JavaScript sequenced-before edges to the matching program-order edges in Armv8-A);
- preserves the observable behaviour:
  preserves reads-byte-from between $Exec_{\text{JS}}$ and $Exec_{\text{ARM}}$.

We give the event-to-event mapping of this translation below; we omit the (unsurprising) details of the mappings on relations of the candidate executions here, but give the full definition in the supplemental material [16]. The event mapping is one-to-one, bearing in mind the transformation discussed above, where the JavaScript RMW event is split in two.

Instructions		**Events**	
JavaScript	**Armv8-A**	**JavaScript**	**Armv8-A**
A-.load	`ldar`	$\mathsf{R_{sc}}$	$\mathrm{R_{acq}}$
A-.store	`stlr`	$\mathsf{W_{sc}}$	$\mathrm{W_{rel}}$
_ = b[k]	`ldr`	$\mathsf{R_{un}}$	R
b[k] = _	`str`	$\mathsf{W_{un}}$	W
A-.exchange	... `ldaxr`/`stlxr` ...	$\mathsf{RMW_{sc}}$	$\mathrm{R_{e\text{-}a}}$ **sb** $\mathrm{W_{e\text{-}r}}$

Our Alloy counter-example search looks for a JavaScript candidate execution $Exec_{\text{JS}}$ and an Armv8-A candidate execution $Exec_{\text{ARM}}$, both well-formed, such that they are related by the translation relation, and $Exec_{\text{ARM}}$ is valid in Armv8-A, but $Exec_{\text{JS}}$ invalid in JavaScript.

5.5.2 Finding counter-examples

For the uncorrected JavaScript model, we would like our search to produce counter-examples similar to Fig 5.11. However, naively searching as described above yields spurious counter-examples. An example is shown in Fig 5.15. This pair of executions satisfies the constraints of our search as specified so far: an invalid JavaScript execution, translation-related to a valid Armv8-A execution. The uncorrected JavaScript here forbids the execution, because it exhibits the forbidden shape of §5.3.2. However, this counter-example is spurious, as a different choice of total-order would make the execution allowed by flipping the edge from (a) to (b). Any program exhibiting this candidate execution will not be a real counter-example, because it will also exhibit the candidate execution with the correct total-order, which is observably equivalent.

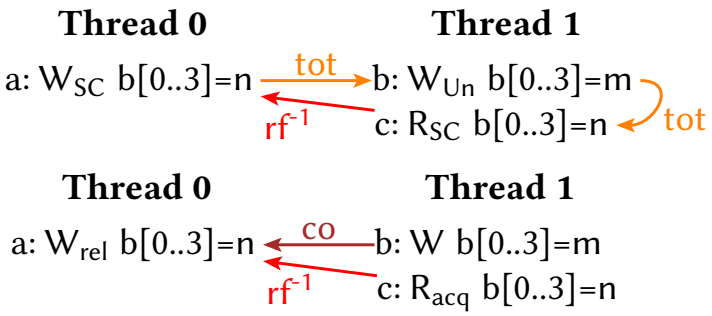


Figure 5.15: False counter-example from naive search.

The problem illustrated by this example is due to the mismatch in the data of Armv8-A and JavaScript candidate executions: assuming a particular Armv8-A execution, the translation relation together with the well-formedness conditions constrains the relations of a corresponding (translation-related) JavaScript execution, except for its (existentially quantified) total-order component. Hence the naive counter-example search will simply pick a "bad" total-order, that is inconsistent with other relations of the JavaScript execution. We are only interested in counter-examples where the JavaScript execution cannot be made valid simply by permuting total-order. Wickerson et al. [165] describe counter-example executions satisfying this requirement as having the *deadness* property.[2]

A way of guaranteeing "good" counter-examples (that are *dead*) would be specifying the search as the question: "does there *exist* a valid Armv8-A execution $Exec_{\text{ARM}}$, such that there *exists* a JavaScript execution $Exec_{\text{JS}}$, that is translation-related to $Exec_{\text{ARM}}$ and such that $Exec_{\text{JS}}$ is invalid in JavaScript *for all* choices of total-order?" Since this Alloy search is computationally infeasible, we use the *syntactic deadness* criterion of Wickerson et al. [165]. This is a syntactic condition on candidate executions that approximates execution deadness in a way that is computationally feasible to check, but which may discard some legitimate counter-examples.

[2]Such executions are "dead" in the sense that they "cannot move around".

For JavaScript, any condition that guarantees that candidate executions differing only in their total-order are required to preserve W_{sc} total-order W_{any} and W_{any} total-order R_{sc} edges, is sufficient to guarantee deadness (we verify this in Coq, based on the model in §5.6). Note in particular that the "counter-example" of Fig. 5.15 does *not* satisfy this condition, as the total-order edge from (a) to (b) can be inverted to create a valid execution. Defining such a search, we successfully find the counter-example in Fig. 5.11.

5.5.3 Bounded compilation scheme correctness

With the JavaScript model fixed as detailed in §5.3, we use Alloy to confirm that no counter-examples exist up to a bound (8 distinct events, 20 locations). This also gives us the opportunity to test proof strategies in preparation for our Coq proof of compilation scheme correctness (§5.6). In that proof, we must show that for any Armv8-A-allowed execution a valid related JavaScript execution exists, which requires constructing a witnessing total-order relation. We model checked our idea for this construction: making total-order some linear extension [166] of $\mathbf{sb} \cup (\mathbf{obs} \cap (L \cup A)^2)$, where $\mathbf{obs} \cap (L \cup A)^2$ is Armv8-a's observed-before relation restricted to release-acquire atomics (see the model definition in Appendix A). With total-order constrained in this way, model checking even without the syntactic deadness approximation shows the absence of compilation scheme counter examples up to the search bound.

5.5.4 SC-DRF search

We are also able to automatically find counter-examples for (the JavaScript model's version of) SC-DRF in the uncorrected model. We use the same search bound, and again we must use our syntactic deadness condition to remove spurious counter-examples. We find the counter-example of Fig. 5.13.

5.6 Coq verification

We mechanise the JavaScript model, as shown in Figs. 5.6 and 5.8, in Coq. See the supplemental material [16] for the precise mechanisation.

The mechanisation is built on top of the existing Intermediate Memory Model (IMM) framework [146], which carries out a number of compilation scheme correctness proofs between different relaxed memory models. Since this work was not carried out in Isabelle/HOL, it is incompatible with our existing mechanisation of sequential WebAssembly (Chapter 3). This is unfortunate, but without the IMM the proofs described below would have been far more challenging and time-consuming to accomplish.

5.6.1 SC-DRF

We first prove that our corrected model is SC-DRF in the sense defined in §5.3.3, mechanising a previous hand-proof by Watt et al. [13]:

Lemma 5.1 (`internal_sc_drf`) *All well-formed, valid, data-race-free executions in the revised JavaScript model are sequentially consistent (taking the model's own definition).*

5.6.2 Compilation scheme correctness

We now prove compilation scheme correctness, from the revised JavaScript model to our Armv8-a model. As mentioned in §5.4, a limitation of this proof is the assumption that all accesses have been generated by typed arrays (i.e. are aligned). This simplifies the proof, since unaligned Arm accesses must be split into separate bytewise events [148].

We build our proof following the style of other proofs in the IMM framework [146]. As in this work, the proof proceeds by defining a "base execution" that is shared between the two models (i.e, intra-thread program order and reads-byte-from), and then showing that, for any such execution, validity in the Armv8-a model implies validity in the JavaScript model. As an intermediate lemma, we must prove that, given an allowed Armv8-A execution, it is possible to construct a witnessing total-order relation for an allowed JavaScript execution. We achieve this proof using the construction we model-checked as part of §5.5.3. The initial model-checking allowed us to rapidly validate possible constructions; it would have been far more time-consuming to come up with a correct construction from scratch.

Lemma 5.2 (`jsmm_compilation`) *The compilation scheme from the revised JavaScript model to (mixed-size) Armv8-A preserves execution validity.*

5.7 Outstanding issues in JavaScript

The JavaScript memory model was explicitly designed to support the compilation of C/C++ to the asm.js subset of JavaScript. It is therefore expected that C/C++'s concurrency primitives map soundly to those of JavaScript. However, there is a counter-example, which highlights an issue in the JavaScript model with the following three inter-related consequences, which will be explained below in further detail:

- Thin-air behaviour
- Invalidity of C++11 compilation scheme
- Weakness of SC-DRF definition

```
x = 0;  y = 0;
Thread 1          Thread 2
if (x == 1)       if (y == 1)
  y = 1;            x = 1;
```

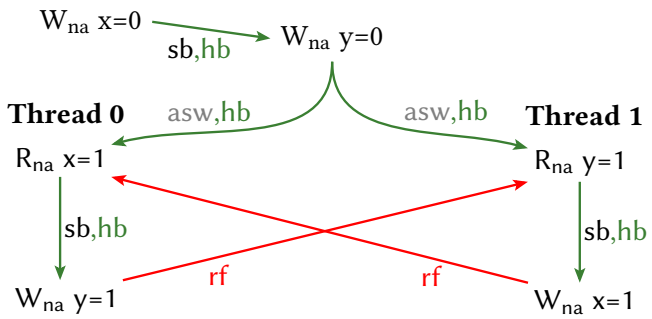


Figure 5.16: A simple C program made up of non-atomic accesses, together with an inconsistent candidate execution in which the program terminates with the outcome x==1, y==1.

```
atomic_init(x, 0);    atomic_init(y, 0);
Thread 1                          Thread 2
if (x.load(memory_order_relaxed) == 1)    if (y.load(memory_order_relaxed) == 1)
  y.store(1, memory_order_relaxed);         x.store(1, memory_order_relaxed);
```

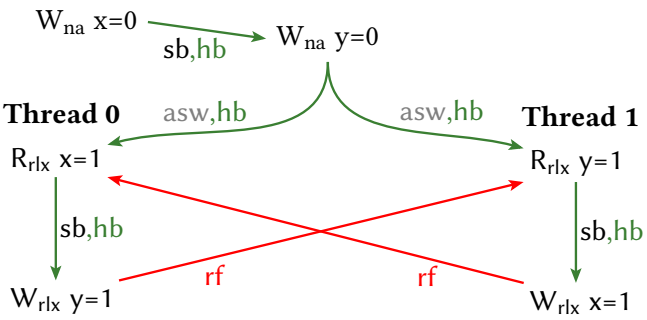


Figure 5.17: The program of Fig. 5.16, except with relaxed atomic accesses. In this example, the execution is consistent.

To explain, we must first give more background on the behaviour of the C++11 memory model. Consider the simple C program of Fig. 5.16, which has been previously discussed by Boehm and Adve [131], and its associated execution exhibiting a *self-satisfying conditional* [134]. Under the classic statement of SC-DRF (see §5.3.3), this program contains no data races, and should therefore exhibit only sequentially-consistent behaviour. Intuitively, because x and y are initially set to 0, the program should terminate without executing the body of either conditional, with the outcome x==0, y==0.

If the candidate execution of Fig. 5.16 were consistent, the program would be spuriously considered to have a data race, based on the C++11 definition (see §5.3.3). To prevent this, C++11 model requires that non-atomics may only read from the most recent write to the same location which *happens-before* themselves. This is known as the *Last Visible Side-Effect* rule. This renders the candidate execution of Fig. 5.16 inconsistent, as desired, since for each read the only write which *happens-before* it is the initializing write of 0. The program is therefore data-race-free under C++11's definition, and has defined behaviour. The Last Visible Side-Effect rule can *only* be included in the C++11 model because any otherwise consistent candidate execution containing a non-atomic read which reads from a write which is *not happens-before* itself must necessarily contain a C++11 data race, and therefore have undefined behaviour.

Unintuitively, if the non-initialisation accesses of Fig. 5.16 are instead relaxed atomics, as depicted in Fig. 5.17, the candidate execution *is* allowed. Because the Last Visible Side-Effect rule no longer applies, and the program is no longer considered to have a C++11 data race because no non-atomics are involved, this undesirable outcome becomes defined behaviour. This deficiency in the model, the existence of so-called *thin-air executions*, is known as *Out-Of-Thin-Air* (OOTA) problem and has received considerable research attention [134, 136, 144, 137, 135]. The Last Visible Side-Effect rule cannot be extended to also constrain relaxed atomics. This would incorrectly forbid a number of concretely observable executions: the whole point of atomic accesses is that they can legitimately occur concurrently in a program without explicit synchronization between them. In fact, it has been shown that, for a model in the style of C++11, expressed purely in terms of axiomatic constraints applied separately to each candidate execution (a *per-candidate-execution* model), there are *no expressible constraints* which discard all OOTA executions while allowing legitimately observable ones, given the desired compilation schemes and compiler optimisations when compiling C/C++ to platform assembly. This result has been succinctly summarised as "the thin-air problem has no per-candidate-execution solution" [134].

As discussed by Lahav et al. [144], the permitted execution of Fig. 5.17 violates an interpretation of the classic SC-DRF property where C++11's relaxed atomics as being like any other non-synchronizing access for the purposes of causing a data race.

While the JavaScript model is heavily based on that of C++11, as discussed above, it

cannot declare data races to be undefined behaviour. JavaScript is thus unable to adopt a Last-Visible Side-Effect constraint like that of C++11. The language instead adopted a "super-relaxed" semantics which straddles a line between C++11 non-atomics and relaxed atomics. Like C++11 relaxed atomics, JavaScript non-atomics allow data races and do not obey a Last Visible Side-Effect condition, although in contrast to C++11 relaxed atomics they obey no per-location-coherency axioms [141]. When the C++11 program of Fig. 5.16 is compiled to JavaScript according to the standard compilation scheme, as depicted in Fig. 5.18, the thin-air execution is permitted.

```
b = new Int8Array(new SharedArrayBuffer(..));
Thread 1          Thread 2
if (b[x] == 1)    if (b[y] == 1)
  b[y] = 1;         b[x] = 1;
```

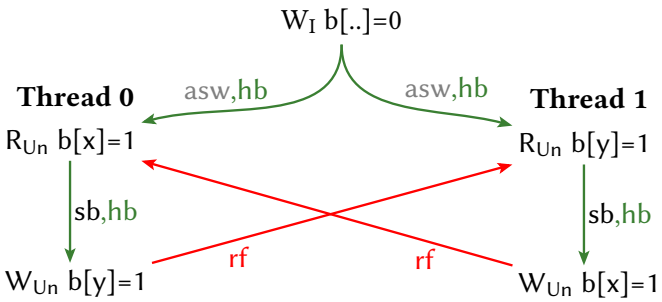


Figure 5.18: The program of Fig. 5.16, as it might be compiled to JavaScript. Variables (assumed here to be chars) become indices into a SharedArrayBuffer. The execution is consistent according to JavaScript's relaxed memory model.

5.7.1 Consequences

Compilation from concurrent C/C++ to JavaScript/WebAssembly is based on an unsound compilation scheme, and due to the characteristics of the OOTA problem, any fix must involve a revised JavaScript model which is not purely per-candidate-execution. This does not imply that any errors will be concretely observable as a result of compilation: C/C++ accesses compiled through JavaScript will be ultimately compiled to platform assembly which is at least as strict as platform assembly generated through direct compilation from C/C++. It is also vanishingly unlikely that JavaScript toolchains implement any re-ordering optimisations which are not already applied to C/C++ by GCC and Clang.

This puts the JavaScript model in the same position as C++11: waiting for a new iteration of the model to be developed which is not purely per-candidate-execution. One might reasonably hope that whatever solution is adopted by C++11 to rid relaxed atomics of their OOTA behaviour can be adapted to describe JavaScript and WebAssembly's

non-atomics in a formally satisfying way. Such models are an active area of research [136, 137, 138, 167, 168]. The model of Paviotti et al. [168] in particular can be phrased as an additional series of conditions on top of a fragment of the existing C++11 model, making it an attractive candidate for fixing the JavaScript model with relatively few changes.

As a related point, this means that the JavaScript specification's statement of SC-DRF is weaker than it should be. Because the thin-air execution of Fig. 5.18 is permitted, the program is considered to exhibit a data race according to the JavaScript definition, and is therefore allowed to exhibit a non-SC behaviour without violating JavaScript's stated version of SC-DRF. For the JavaScript model to satisfy the classic statement of SC-DRF, this program should not exhibit a data race, and should always be sequentially consistent. Clearly in any real implementation, the non-SC execution is not allowed, but as discussed above there is no way to correct this issue with the current per-candidate-execution model.

5.8 Extensions for WebAssembly

WebAssembly's core concurrency primitives are identical to those of JavaScript (see Fig. 5.2). This was a deliberate design decision made in order to ensure their seamless interoperability. Given this, we have to be pragmatic when designing WebAssembly's relaxed memory model. Many of its behaviours are fixed by the existing JavaScript model. The only delta between JavaScript and WebAssembly's concurrency capabilities is the latter's addition of the **memory.grow** instruction, which allows the shared buffer to dynamically grow in size. Therefore, the WebAssembly-specific parts of the model focus on defining this behaviour. During the informal design of WebAssembly's concurrency extension, it was intended for memory growth to have a sequentially consistent semantics [139].

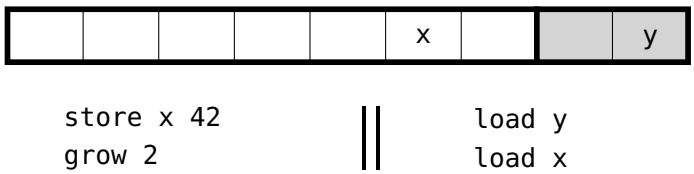


Figure 5.19: Psuedocode representing a race caused by memory growth. Index `y` is out-of-bounds before the `grow`, but in-bounds after.

However, we observe that real-world compilation schemes have a much weaker semantics. In particular, consider the abstracted example of Fig. 5.19. This is a standard Message-Passing (MP) test, except the "flag" indicating the message is ready to be read is replaced with a memory growth and implicit bounds check. The memory location y is outside the bounds of memory before the growth of the first thread, but in-bounds afterwards. In a sequentially consistent model, if the read of y in the second thread succeeds without a out-of-bounds trap, then the `grow` of the first thread must have executed, and therefore

the previous write of 42 to x must have executed. Therefore, the subsequent read of x in the second thread must read 42.

To determine WebAssembly's model of concurrent memory growth, the question must be asked: is a weaker outcome possible in a realistic implementation? For example, can the read of x in the second thread read 0? To determine whether this is possible, we examine two desirable compilation schemes for memory growth.

5.8.1 Implementation of memory.grow

5.8.1.1 Explicit

On mobile and 32-bit devices, a naive implementation is often used, where every access is compiled with an explicit bounds check. The shared memory is initially allocated with extra space beyond what is programmatically accessible. A "length" value is kept track of (like a global variable). Regular WebAssembly memory accesses are implemented by first explicitly bounds-checking against this length value using a regular platform assembly load. The **memory.grow** instruction is implemented as an atomic increment of the length value. If the length would be incremented beyond the pre-allocated memory size, the operation fails (remember that WebAssembly's semantics allows **memory.grow** to fail non-deterministically).

Fig. 5.20 represents the program of Fig. 5.19 implemented in this way. Bounds checks are performed as regular accesses. The Armv8-A memory model allows this program to exhibit a relaxed behaviour, where the bounds check of `y` in the second thread succeeds, but the read of `x` observes 0. This demonstrates that WebAssembly's memory growth semantics cannot be sequentially consistent. Moreover, it is currently possible to optimise away such bound checks altogether. For example, if it were known that `x <= y`, then the final bounds check of the second thread could be optimised away. This also provides intuition that the bounds checks themselves cannot be relied on as a source of ordering constraints/synchronisation.

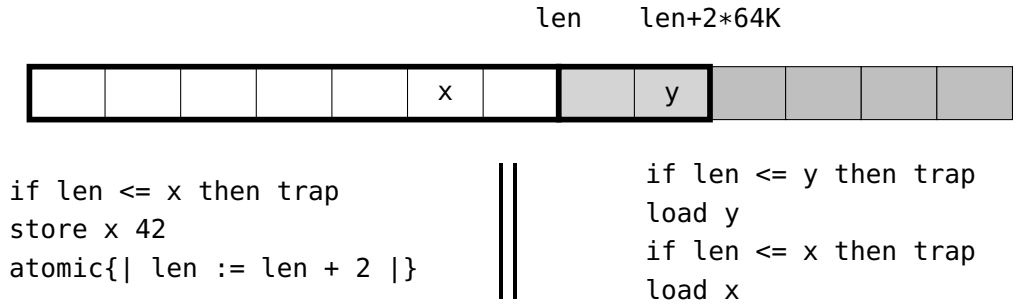
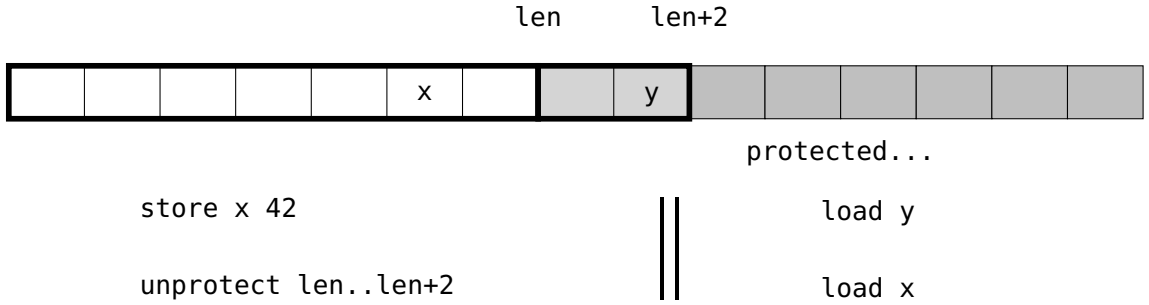


Figure 5.20: Psudocode representing an explicit bounds checking implementation strategy for Fig. 5.19.

5.8.1.2 Trap handler

Clearly requiring each WebAssembly memory access to perform an explicit bounds check is not ideal performance-wise. A more optimised implementation exists for 64-bit machines, taking advantage of OS-level *page protection*. This approach is used by the Web engines V8 (Chrome) and SpiderMonkey (Firefox). WebAssembly memories are accessed using 32-bit indices. When a shared memory is allocated on a 64-bit machine, it is possible for the whole 32-bit virtual address space of the memory to be reserved[3]. Address ranges beyond the bounds of the current programmatically-accessible memory are read/write-protected. WebAssembly accesses are then implemented as bare assembly accesses. If an access occurs beyond the current WebAssembly memory bound, a page protection fault occurs. An installed *trap handler* catches this fault and converts it back into a WebAssembly-level **trap**. This implementation has the advantage that no programmatic bounds checks are required. However, this approach is only possible on 64-bit platforms.

Fig. 5.21 represents this implementation strategy. In contrast with Fig. 5.20, memory accesses are not guarded by bounds checks. Instead, accesses beyond the bounds of the memory will cause a page protection fault. We are currently not able to reason formally about this implementation, as research on relaxed behaviours of the page table/exceptions/the OS has not yet reached the necessary level of sophistication. However, it does provide further intuition that the "bounds checks" of memory accesses, even atomic ones, cannot be expected to be implemented in a way that supports strong ordering constraints in the model.




```
protection_trap_handler:
  clean up, create wasm-level trap
```

Figure 5.21: Psudocode representing a bounds checking implementation strategy for Fig. 5.19 based on page protection.

[3]Note that reserving virtual memory is an inexpensive operation which does not *commit* the addresses into main memory (RAM).

5.8.2 The model

The initial informal design of WebAssembly threads intended for memory growth to have sequentially consistent behaviour [139]. However, it quickly became clear to us when writing the formal model that this would not be possible, given the desired implementation schemes described above. Because memory access bounds checks are implemented either using bare assembly accesses, or elided entirely in implementations, we are naturally led to modelling them as non-atomic accesses. Loads and stores are modelled as carrying out an additional Unordered access to a distinguished *length* location. The **memory.grow** instruction is modelled as an SC atomic RMW on the location, while **memory.size** is modelled as an SC atomic read. This means that explicitly checking/changing the length of memory through **memory.size**/**grow** will still cause a synchronisation (with the last **grow**), and implementations need to ensure they perform the correct barriers to enforce this. This formalisation essentially blesses the explicit non-atomic bounds checking approach of §5.8.1.1 within the formal model. As mentioned above, we cannot currently formally verify the correctness of the trap handler approach, although the lack of synchronisation on access bounds checks fits our expectations.

The execution of a memory instruction now involves multiple concurrent operations in a single step. To model, this, we generalise JavaScript's events so that a single event can now contain multiple *actions* over separate locations (Fig. 5.22). For example, an **atomic.store** will generate an event with two actions: one reading from the *length* location with un consistency, and one reading from the buffer with sc consistency.

The execution witness must be similarly generalised so that reads-byte-from records the *loc* on which a read is taking place; either a regular read of memory denoted by data, or a read of the memory length denoted by len. The derived relations reads-from and synchronizes-with are also parameterised in this way – definitions are given in Fig. 5.23.

$$
\begin{array}{lcl}
\mathit{loc} & ::= & \mathsf{len} \mid \mathsf{data} \\
\mathit{action} & ::= & \{\ \mathrm{ord} :: \mathit{mode},\ \ \mathrm{reads} :: \mathit{list\ byte}, \\
 & & \ \ \mathrm{writes} :: \mathit{list\ byte},\ \ \mathrm{index} :: \mathit{nat}, \\
 & & \ \ \mathrm{tearfree} :: \mathit{bool}\ \ \} \\
\mathit{event} & ::= & \{\ \mathrm{action} :: \mathit{loc} \rightharpoonup \mathit{action}\ \}_{\mathit{id}}
\end{array}
$$

Figure 5.22: WebAssembly events. Compare to the JavaScript events of Fig. 5.4.

$$witness \quad ::= \quad \{\ \text{reads-byte-from} \mathbin{/\!/} \mathbf{rbf} :: set\ (loc \times nat \times event \times event),\ \text{total-order} \mathbin{/\!/} \mathbf{tot} :: set\ (event \times event)\ \}$$

$$\text{range}^{l}_{r|w}(E) \triangleq \text{range}_{r|w}(E.\text{action}(l)) \qquad E.\text{ord}^{l} \triangleq E.\text{action}(l).\text{ord} \qquad \text{reads-from} \mathbin{/\!/} \mathbf{rf} \triangleq \{\langle l, A, B\rangle \mid \exists k.\, \langle l, k, A, B\rangle \in \mathbf{rbf}\}$$

$$\text{synchronizes-with} \mathbin{/\!/} \mathbf{sw} \triangleq \left\{ \langle l, A, B\rangle \;\middle|\; \begin{array}{l} \langle l, A, B\rangle \in \mathbf{rf} \wedge B.\text{ord}^{l} = \mathsf{SeqCst} \wedge \\ \left(\begin{array}{l} \left(\text{range}^{l}_{w}(A) = \text{range}^{l}_{r}(B) \ \wedge A.\text{ord}^{l} = \mathsf{SeqCst}\right) \vee \\ \left(\forall C.\ \langle l, C, B\rangle \in \mathbf{rf} \implies C.\text{ord}^{l} = \mathsf{Init}\right) \end{array}\right) \end{array} \right\}$$

$$\text{happens-before} \mathbin{/\!/} \mathbf{hb} \triangleq \left(\begin{array}{l} \mathbf{sb} \cup \mathbf{sw}(\mathsf{len}) \cup \mathbf{sw}(\mathsf{data}) \cup \mathbf{asw} \cup \\ \{\ \langle A, B\rangle \mid A.\text{ord}^{l} = \mathsf{Init} \wedge \text{overlap}^{l}(A, B)\ \} \end{array}\right)^{+}$$

Figure 5.23: WebAssembly execution witness and derived relations

5.9 Related work

5.9.1 Language-level relaxed memory

Many of the conventions used by modern source-language relaxed memory models are drawn from the work of Adve and Hill [130], which first defined the SC-DRF correctness condition. Subsequently, several attempts were made to define a memory model for Java [169, 170, 171], but the ultimately adopted model was soon shown not to match implementation reality [172]. The C++11 model [131, 141] aimed to avoid known deficiencies in the Java model – by declaring all non-atomic data races as undefined behaviour, many unfortunate corner-cases of the Java model could be avoided [173, 172]. However, such data races are often necessary to implement efficient concurrent algorithms, so C++11 introduced a family of so-called *low-level atomics* with different ordering behaviours which were allowed to race without causing undefined behaviour, but were intended to be more strongly ordered than non-atomics. The semantics of some of these atomics caused significant issues: the `consume` ordering was essentially abandoned due to implementation difficulty [174], and the specification of the `relaxed` ordering led to the thin-air problem, one of the most notorious issues still plaguing the field [134].

Nevertheless, the C++11 model was a seminal work, and has been adapted for several other languages (including JavaScript, as detailed in this chapter). The OCaml memory model [132] adapts C++11 but avoids its issues by committing to a more expensive compilation scheme and fewer compiler optimisations for mutable state. This is a trade-off that OCaml can make because such mutable state occurs comparatively rarely in its programs. A new iteration of the Java model [133] is also based on C++11, although like JavaScript it attempts to give defined behaviour to racing non-atomics, resulting in thin-air issues analogous to those discussed in §5.7.

Another strain of research has attempted to define a new model for the accesses of C++11 which avoids thin-air behaviours. The RC11 model [144] prevents thin-air behaviours in relaxed atomics at the cost of requiring the insertion of extra instructions after each load (either a never-taken branch to create syntactic control dependencies, or a barrier) on certain architectures. The promising semantics [136, 138] aims to avoid the issues with axiomatic models by defining an operational one, although it has recently been discovered to exhibit its own thin-air behaviours [167]. More recent work has proposed models which compare semantic dependencies across multiple executions to permit "dependency breaking" optimisations while disallowing thin-air behaviours [167, 168].

The (uncorrected) JavaScript model has been previously modelled in Alloy by Mattarei et al. [175]. This work found definition-level errors in an earlier draft of the model (such as the model allowing an RMW event to read from itself). Their EMME tool used this Alloy model to generate the allowed executions for small provided programs. Their work did not concentrate on a qualitative assessment of the model's properties and therefore did not identify the issues we discuss above. While we also use Alloy, we found it easier to define our own model from scratch to fit the approach of MemAlloy [165], as opposed to adapting their model for this purpose.

Based on our corrected model, some initial work has been done in verifying JavaScript compiler optimisations [176].

Almost no existing work deals with mixed-size behaviours at the source-language-level. Flur et al. [148] propose an extension to the C++11 model to handle mixed-size non-atomics and give a sketch proof of compilation correctness to their mixed-size POWER architecture model.

5.9.2 Architecture-level relaxed memory and compilation

Language-level relaxed memory models abstract both the action of the compiler, and the underlying relaxed behaviour of the target architecture. The relaxed behaviour allowed at the architecture level itself has been extensively studied [177, 178, 179, 180, 145, 181, 182, 183, 184, 185, 186, 187, 159, 188, 189, 190, 191, 192]

In particular, the especially relaxed models of the modern Armv8-A and POWER architectures are recent subjects of active research and industry iteration [159, 153, 142, 151]. Some work has been done on the mixed-size behaviours of these architectures [148], and of the instruction fetch semantics of Armv8-A [193]. Since the publication of our Armv8-A axiomatic model, research on an Arm-official mixed-size axiomatic model has been published [164] which includes specific comparisons to our work and Flat.

Hand-proofs of compilation scheme correctness between a language-level and architecture-level relaxed memory model are often attempted, but sometimes stumble on edge-cases. For example a purported proof of compilation scheme correctness from C++11 to

POWER [140] was later found to false [194, 144]. We ourselves originally claimed to provide a proof sketch of compilation scheme correctness from C++11 to WebAssembly [13], despite the existence of a clear counter-example (§5.7). The paper has now been corrected [14]). Mechanised verification of compilation scheme correctness is far less common. CompCertTSO [195] builds on CompCert [54] to mechanise a proof of compilation correctness of a significant subset of the C language to a TSO model. The Intermediate Memory Model (IMM) framework defines a unifying mechanised proof infrastructure which connects multiple source-level memory models with underlying architectural models [146]. We build on it to prove various results about the JavaScript model, as discussed in §5.6.2. The MemAlloy project [165] conducts exhaustive counter-example searches of compilation schemes up to a bound for models expressible in the `cat` format [152], and has succeeded in automatically reproducing several compilation scheme correctness bugs which were originally found by hand with great effort.

5.10 Future work

Clearly, as discussed in §5.7.1, the underlying thin-air issues of the JavaScript model need to be fixed. The approach of Paviotti et al. [168] is under consideration for adoption into the C++ specification. If successful, their treatment of C++ relaxed atomics will likely provide a blueprint for a revised definition of JavaScript's non-atomics, to address the current issues discussed in §5.7.

We would also like to be able to verify the correctness of the trap-handler WebAssembly bounds checking implementation (Fig. 5.21). As discussed above, this would require more sophisticated models of exceptions, page tables, and related OS behaviour. Some initial steps have been taken in these areas [193, 196].

We still lack a published mixed-size relaxed memory model of x86 accesses. This will be a necessary step for proving compilation correctness from the full JavaScript/WebAssembly model to that platform.

It is likely that performance pressures will lead WebAssembly to specify additional (more relaxed) low-level atomics in the style of C++11. It is crucial that any short-term additions to the language still leave the door open for future improvements to the relaxed memory model.

In future, it would be valuable to extend the Isabelle/HOL mechanisation of sequential WebAssembly (Chapter 3) with our relaxed memory model.

Chapter 6

Conclusion

The World Wide Web is a ubiquitous and essential part of modern-day life, and great care must therefore be taken in designing the technologies underpinning it. The fact that WebAssembly has been designed so closely around a formal specification demonstrates a growing appreciation in industry of the value of this approach.

While it is admirable that WebAssembly's designers were able to produce a pen-and-paper formal semantics, my work shows that mechanisation offers a level of confidence in the correctness of the specification which cannot be achieved through purely hand-written definitions. Because I was able to offer my input during the drafting of the published specification, a number of errors and omissions were discovered at an early stage. My mechanised proof of WebAssembly's type soundness property, in Isabelle/HOL, is now cited by the official specification as evidence of the correctness of the language's design. WebAssembly continues to evolve, and my work on CT-Wasm shows that my mechanisation can be used to motivate the correctness of proposed extensions to the language.

The in-progress threads feature, which introduces relaxed memory behaviours to WebAssembly, represents a major complication for the language. My work on the inter-related JavaScript and WebAssembly relaxed memory models ultimately aims to provide assurance that the formal specification of the WebAssembly threads feature is correct. While significant progress has been made through correcting errors in the JavaScript model, and pinning down the unique aspects of the WebAssembly model, there is still much to be done before this goal is reached.

Bibliography

[1] Allen Wirfs-Brock and Brendan Eich. JavaScript: The first 20 years. This is the authors HOPL-4 preprint. The definitive Version of Record to be published in Proceedings of the ACM on Programming Languages, June 2020, https://doi.org/10.1145/3386327., March 2020. URL https://doi.org/10.1145/3386327.

[2] Rawn Shah. Bending over backward to make JavaScript work on 14 platforms. JavaWorld 1, 2, 1996. URL http://www.javaworld.com/javaworld/jw-04-1996/jw-04-jsinterview.html. https://web.archive.org/web/19970104122216/http://www.javaworld.com/javaworld/jw-04-1996/jw-04-jsinterview.html.

[3] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, January 2010. ISSN 0001-0782. doi: 10.1145/1629175.1629203. URL https://doi.org/10.1145/1629175.1629203.

[4] Alon Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, page 301–312, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309424. doi: 10.1145/2048147.2048224. URL https://doi.org/10.1145/2048147.2048224.

[5] Alon Zakai. Issue 4392: Turbofan/asm.js memory not immediately released, 2015. URL https://bugs.chromium.org/p/v8/issues/detail?id=4392#c_ts1442832710. to archive.

[6] ECMA TC39. SIMD.js, 2018. URL https://github.com/tc39/ecmascript_simd.

[7] Brad Nelson. Issue 6020: [wasm] Implement WebAssembly SIMD prototype (comment #3), 2017. URL https://bugs.chromium.org/p/v8/issues/detail?id=6020&desc=2#c3. to archive.

[8] JF Bastien. Going public launch bug, 2015. URL https://github.com/WebAssembly/design/issues/150.

[9] Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 53–65, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355865. doi: 10.1145/3167082. URL https://doi.org/10.1145/3167082.

[10] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290390. URL https://doi.org/10.1145/3290390.

[11] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002. ISSN 1049-331X. doi: 10.1145/505145.505149. URL http://doi.acm.org/10.1145/505145.505149.

[12] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *EUROCAL '85*, pages 151–184, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. ISBN 978-3-540-39684-0.

[13] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. Weakening WebAssembly. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360559. URL https://doi.org/10.1145/3360559.

[14] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. Corrigendum: Weakening WebAssembly. December 2020. URL https://dl.acm.org/doi/10.1145/3360559.

[15] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. Repairing and mechanising the JavaScript relaxed memory model. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 346–361, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385973. URL https://doi.org/10.1145/3385412.3385973.

[16] Conrad Watt. Thesis supplemental materials, 2021. URL https://github.com/conrad-watt/wasm-thesis-aux.

[17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to

speed with WebAssembly. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017. ISBN 978-1-4503-4988-8.

[18] Lin Clark. Making WebAssembly even faster: Firefox's new streaming and tiering compiler, 2018. URL https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler/.

[19] WebAssembly Community Group. Why create a new standard when there is already asm.js?, 2018. URL https://webassembly.org/docs/faq/#why-create-a-new-standard-when-there-is-already-asmjs.

[20] WebAssembly Community Group. WebAssembly W3C Process, 2020. URL https://github.com/WebAssembly/meetings/blob/master/process/phases.md.

[21] Mozilla Foundation. What is web compatibility? URL https://webcompat.com/about.

[22] Xavier Leroy. Java bytecode verification: An overview. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 265–285, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44585-2.

[23] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[24] Ori Bernstein. Please support arbitrary labels and gotos., 2016. URL https://github.com/WebAssembly/design/issues/796.

[25] WebAssembly Community Group. reference-types, 2018. URL https://github.com/WebAssembly/reference-types.

[26] WebAssembly Working Group. Webassembly core specification, 2019. URL https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/.

[27] WebAssembly Community Group. WebAssembly, 2021. URL https://webassembly.github.io/spec/core.

[28] M. Clint and C. A. R. Hoare. Program proving: Jumps and functions. *Acta Informatica*, 1(3):214–224, Sep 1972. ISSN 1432-0525. doi: 10.1007/BF00288686. URL https://doi.org/10.1007/BF00288686.

[29] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4), January 2008. ISSN 1544-3566. doi: 10.1145/1328195.1328197. URL https://doi.org/10.1145/1328195.1328197.

[30] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994. ISSN 0890-5401. doi: 10.1006/inco.1994.1093. URL https://doi.org/10.1006/inco.1994.1093.

[31] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091.

[32] WebAssembly Community Group. Soundness, 2018. URL https://webassembly.github.io/spec/core/appendix/properties.html.

[33] Ben Titzer. Thoughts on improving compilation model (comment), 2020. URL https://github.com/WebAssembly/design/issues/1375#issuecomment-693973769.

[34] WebAssembly Community Group. Instantiation, 2020. URL https://webassembly.github.io/spec/core/exec/modules.html#instantiation.

[35] L. C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5 (3):363–397, September 1989. ISSN 0168-7433. doi: 10.1007/BF00248324. URL https://doi.org/10.1007/BF00248324.

[36] Conrad Watt. WasmCert-Isabelle, 2020. URL https://github.com/WasmCert/WasmCert-Isabelle.

[37] Tobias Nipkov. Re: [isabelle] dependent types, 2010. URL https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2010-April/msg00008.html.

[38] Lukas Bulwahn, Fakultät Informatik, Der Technischen Universität München, Diplomarbeit In Informatik, Supervisor Prof, Tobias Nipkow, and Ph. D. Predicates in Isabelle/HOL, 2009.

[39] M. Broy and M. Wirsing. On the algebraic specification of nondeterministic programming languages. In Egidio Astesiano and Corrado Böhm, editors, *CAAP '81*, pages 162–179, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg. ISBN 978-3-540-38716-9.

[40] Ross Tate. Proposal: Add type for unreachable states, 2020. URL https://github.com/WebAssembly/design/issues/1379.

[41] Conrad Watt and Ross Tate. Relaxed dead code validation, 2020. URL https://github.com/WebAssembly/relaxed-dead-code-validation/.

[42] Florian Haftmann and Lukas Bulwahn. Code generation from Isabelle/HOL theories, 2007.

[43] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of RunST. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158152. URL https://doi.org/10.1145/3158152.

[44] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 87–100, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325448. doi: 10.1145/2535838.2535876. URL https://doi.org/10.1145/2535838.2535876.

[45] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, pages 126–150, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14107-2.

[46] Michael Norrish. C formalised in HOL. Technical report, 1998.

[47] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, July 2006. ISSN 0164-0925. doi: 10.1145/1146809.1146811. URL https://doi.org/10.1145/1146809.1146811.

[48] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 173–184, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 1595935754. doi: 10.1145/1190216.1190245. URL https://doi.org/10.1145/1190216.1190245.

[49] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Automated Deduction — CADE-16*, pages 202–206, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48660-2.

[50] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325448. doi: 10.1145/2535838.2535841. URL https://doi.org/10.1145/2535838.2535841.

[51] Yong Kiam Tan, Scott Owens, and Ramana Kumar. A verified type system for CakeML. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450342735. doi: 10.1145/2897336.2897344. URL https://doi.org/10.1145/2897336.2897344.

[52] Scott Owens. A sound semantics for OCamllight. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems*, ESOP'08/ETAPS'08, page 1–15, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787380.

[53] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52 (7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

[54] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems*. SEE, 2016. URL http://xavierleroy.org/publi/erts2016_compcert.pdf.

[55] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 1–15, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342612. doi: 10.1145/2908080.2908081. URL https://doi.org/10.1145/2908080.2908081.

[56] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 175–188, New York, NY, USA, September 2014. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628143.

[57] Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. Cerberus-bmc: A principled reference semantics and exploration tool for concurrent and sequential c. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 387–397, Cham, 2019. Springer International Publishing. ISBN 978-3-030-25540-4.

[58] Grigore Roșu and Traian Florin Șerbănuță. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397 – 434, 2010. ISSN 1567-8326. doi: https://doi.org/10.1016/j.jlap.2010.03.012.

URL http://www.sciencedirect.com/science/article/pii/S1567832610000160. Membrane computing and programming.

[59] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 533–544, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310833. doi: 10.1145/2103656.2103719. URL https://doi.org/10.1145/2103656.2103719.

[60] Denis Bogdanas and Grigore Roşu. K-Java: A complete semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 445–456, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2676982. URL https://doi.org/10.1145/2676726.2676982.

[61] Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 346–356, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737991. URL https://doi.org/10.1145/2737924.2737991.

[62] Daniele Filaretti and Sergio Maffeis. An executable formal semantics of PHP. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 567–592, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44202-9.

[63] A. Stefanescu. MatchC: A matching logic reachability verifier using the K framework. *Electron. Notes Theor. Comput. Sci.*, 304:183–198, 2014.

[64] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364416. doi: 10.1145/3236950.3236956. URL https://doi.org/10.1145/3236950.3236956.

[65] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part i: A multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 927–942, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386014. URL https://doi.org/10.1145/3385412.3386014.

[66] John Renner, Sunjay Cauligi, and Deian Stefan. Constant-time webassembly. In *Principles of Secure Compilation*, 2018.

[67] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of Advances in Cryptology*. Springer, 1996.

[68] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5), 2005.

[69] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[70] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference*. Springer, 2006.

[71] Daniel J Bernstein. The Salsa20 family of stream ciphers. In *New stream cipher designs*. Springer, 2008.

[72] Daniel J Bernstein. The Poly1305-AES message-authentication code. In *Proceedings of the International Workshop on Fast Software Encryption*. Springer, 2005.

[73] Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *Proceedings of the International Workshop on Public Key Cryptography*. Springer, 2006.

[74] Cryptography Coding Standard. Coding rules, 2016. URL `https://cryptocoding.net/index.php/Coding_rules`.

[75] Thomas Pornin. Why constant-time crypto?, 2017. URL `https://www.bearssl.org/constanttime.html`.

[76] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2016. ISBN 978-1-931971-32-4.

[77] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time mee-cbc. In *Revised Selected Papers of the International Conference on Fast Software Encryption*. Springer-Verlag New York, Inc., 2016. ISBN 978-3-662-52992-8.

[78] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Proceedings of the European Symposium on Research in Computer Security*. Springer International Publishing, 2017.

[79] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[80] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[81] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2017. ISBN 978-1-931971-40-9.

[82] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. FaCT: A flexible, constant-time programming language. In *Proceedings of the IEEE Cybersecurity Development Conference*. IEEE, 2017.

[83] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[84] Michele Boreale. Quantifying information leakage in process calculi. *Information and Computation*, 207(6), 2009. ISSN 0890-5401.

[85] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002.

[86] Benjamin Beurdouche. Verified cryptography for Firefox 57, 2017. URL `https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/`.

[87] D. Page. A note on side-channels resulting from dynamic compilation. In *Cryptology ePrint Archive*, 2006. URL `https://eprint.iacr.org/2006/349`.

[88] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015. ISBN 978-1-4503-3832-5.

[89] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015. ISBN 978-1-4503-3832-5.

[90] Ryan Sleevi. W3C Web Crypto API Update. IETF 86, 2013. URL `https://datatracker.ietf.org/meeting/86/materials/slides-86-saag-5`.

[91] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Defensive javascript. In *Foundations of Security Analysis and Design VII*. Springer, 2014.

[92] Open Whisper Systems. Signal protocol library for JavaScript, 2016. URL `https://github.com/signalapp/libsignal-protocol-javascript`.

[93] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *Proceedings of the IEEE European Symposium on Security and Privacy*. IEEE Computer Society, 2017.

[94] Dmitry Chestnykh. TweetNaCl.js, 2016. URL `https://www.npmjs.com/package/tweetnacl`.

[95] Daniel Cousens. pbkdf2, 2014. URL `https://www.npmjs.com/package/pbkdf2`.

[96] Fedor Indutny. Elliptic, 2014. URL `https://www.npmjs.com/package/elliptic`.

[97] Dominic Tarr. crypto-browserify, 2013. URL `https://www.npmjs.com/package/crypto-browserify`.

[98] Harry Halpin. The W3C web cryptography API: Design and issues. In *Proceedings of the International Workshop on Web APIs and RESTful design*, 2014.

[99] Node.js Foundation. Node.js, 2018. URL `https://nodejs.org/docs/latest-v10.x/api/crypto.html`.

[100] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014. ISBN 978-1-4503-2957-6.

[101] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe Haskell. In *ACM SIGPLAN Notices*, volume 47. ACM, 2012.

[102] Dominik Strohmeier and Peter Dolanjski. Comparing browser page load time: An introduction to methodology, 2017. URL `https://hacks.mozilla.org/2017/11/comparing-browser-page-load-time-an-introduction-to-methodology/`.

[103] Amos Ndegwa. What is page load time?, 2016. URL `https://www.maxcdn.com/one/visual-glossary/page-load-time/`.

[104] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015. ISBN 978-1-4673-6949-7.

[105] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2017.

[106] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.

[107] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2005, 2001.

[108] Andrew C Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1999.

[109] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1), 2003.

[110] C. Disselkoen, R. Jagadeesan, A. Jeffrey, and J. Riely. The code that never ran: Modeling attacks on speculative evaluation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1238–1255, 2019. doi: 10.1109/SP.2019.00047.

[111] R. McIlroy, J. Sevcík, Tobias Tebbi, Ben L. Titzer, and T. Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *ArXiv*, abs/1902.05178, 2019.

[112] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3), 1996. ISSN 0926-227X.

[113] Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Proving concurrent noninterference. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*. Springer-Verlag, 2012. ISBN 978-3-642-35307-9.

[114] Mark Randolph and William Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2), 2020.

[115] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2017.

[116] Benedikt Meurer and Mathias Bynens. The story of a V8 performance cliff in React, 2019. URL `https://v8.dev/blog/react-cliff`.

[117] Daniel J Bernstein. Writing high-speed software, 2007. URL `http://cr.yp.to/qhasm.html`.

[118] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010.

[119] K. Rustan M. Leino. This is boogie 2. June 2008. URL `https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/`.

[120] Project Everest. HACL*, a formally verified cryptographic library written in F*, 2018. URL `https://github.com/project-everest/hacl-star`.

[121] Manuel Barbosa, Andrew Moss, Dan Page, Nuno F. Rodrigues, and Paulo F. Silva. Type checking cryptography implementations. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*. Springer Berlin Heidelberg, 2012.

[122] Manuel Barbosa, David Castro, and Paulo F. Silva. Compiling CAO: From cryptographic specifications to C implementations. In Martín Abadi and Steve Kremer, editors, *Proceedings of Principles of Security and Trust*. Springer Berlin Heidelberg, 2014.

[123] Galois. Cryptol: The language of cryptography. `https://cryptol.net/files/ProgrammingCryptol.pdf`, 2016.

[124] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2006. ISSN 0733-8716.

[125] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Provably secure compilation of side-channel countermeasures. Cryptology ePrint Archive, Report 2017/1233, 2017. `https://eprint.iacr.org/2017/1233`.

[126] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *Proceedings of the IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2018.

[127] Conrad Watt. (memory model, wait/notify) Atomics.wait/notify non-SC behaviour, what is expected?, 2019. https://github.com/tc39/ecma262/issues/1680.

[128] Conrad Watt. Normative: fix axiomatic model to be SC-DRF, and allow ARMv8 compilation, 2019. https://github.com/tc39/ecma262/pull/1511.

[129] ECMA TC39. Spec: JavaScript Shared Memory, Atomics, and Locks, 2015. https://github.com/tc39/ecmascript_sharedmem/blob/master/historical/Spec_JavaScriptSharedMemoryAtomicsandLocks.pdf.

[130] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, page 2–14, New York, NY, USA, 1990. Association for Computing Machinery. ISBN 0897913663. doi: 10.1145/325164.325100. URL https://doi.org/10.1145/325164.325100.

[131] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 68–78, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375591. URL https://doi.org/10.1145/1375581.1375591.

[132] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. Bounding data races in space and time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 242–255, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192421. URL https://doi.org/10.1145/3192366.3192421.

[133] John Bender and Jens Palsberg. A formalization of java's concurrent access modes. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360568. URL https://doi.org/10.1145/3360568.

[134] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In Jan Vitek, editor, *Programming Languages and Systems*, pages 283–307, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46669-8.

[135] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329170. doi: 10.1145/2618128.2618134. URL https://doi.org/10.1145/2618128.2618134.

[136] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 175–189, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009850. URL http://doi.acm.org/10.1145/3009837.3009850.

[137] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 622–633, January 2016. doi: 10.1145/2837614.2837616.

[138] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0: Global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 362–376, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386010. URL https://doi.org/10.1145/3385412.3386010.

[139] WebAssembly Community Group. threads, 2020. URL https://github.com/WebAssembly/threads.

[140] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 509–520, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310833. doi: 10.1145/2103656.2103717. URL https://doi.org/10.1145/2103656.2103717.

[141] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 55–66, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926394. URL https://doi.org/10.1145/1926385.1926394.

[142] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Proceedings of the 22nd International Conference on Computer Aided Verification, LNCS 6174*, pages 258–272, 2010. doi: http://dx.doi.org/10.1007/978-3-642-14295-6_25.

[143] Conrad Watt. (memory model, wait/notify) atomics.wait/notify non-sc behaviour, what is expected?, 2019. URL https://github.com/tc39/ecma262/issues/1680.

[144] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 618–632, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062352. URL http://doi.acm.org/10.1145/3062341.3062352.

[145] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, page 15–26, New York, NY, USA, 1990. Association for Computing Machinery. ISBN 0897913663. doi: 10.1145/325164.325102. URL https://doi.org/10.1145/325164.325102.

[146] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290382. URL https://doi.org/10.1145/3290382.

[147] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL*, 2(POPL):19:1–19:29, 2018. doi: 10.1145/3158107. URL https://doi.org/10.1145/3158107.

[148] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-Size Concurrency: ARM, POWER, C/C++11, and SC. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 429–442, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346603. doi: 10.1145/3009837.3009839. URL https://doi.org/10.1145/3009837.3009839.

[149] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186, 2011. doi: 10.1145/1993498.1993520. URL https://doi.org/10.1145/1993498.1993520.

[150] Will Deacon. The ARMv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (accessed 2019-07-01), 2016.

[151] Arm Limited. Memory model tool, 2020. URL https://developer.arm.com/architectures/cpu-architecture/a-profile/memory-model-tool.

[152] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014. ISSN 0164-0925. doi: 10.1145/2627752. URL http://doi.acm.org/10.1145/2627752.

[153] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 608–621, 2016. doi: 10.1145/2837614.2837615. URL https://doi.org/10.1145/2837614.2837615.

[154] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 635–646, 2015. doi: 10.1145/2830772.2830775. URL https://doi.org/10.1145/2830772.2830775.

[155] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models, 2012. URL http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

[156] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 311–322, 2012. doi: 10.1145/2254064.2254102. URL https://doi.org/10.1145/2254064.2254102.

[157] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 495–512, 2012. doi: 10.1007/978-3-642-31424-7_36. URL https://doi.org/10.1007/978-3-642-31424-7_36.

[158] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 41–44, 2011. doi: 10.1007/978-3-642-19835-9_5. URL https://doi.org/10.1007/978-3-642-19835-9_5.

[159] Jade Alglave, Anthony C. J. Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and ARM multiprocessor machine code. In *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*, pages 13–24, 2009. doi: 10.1145/1481839.1481842. URL https://doi.org/10.1145/1481839.1481842.

[160] Nathan Chong and Samin Ishtiaq. Reasoning about the ARM weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), Seattle, Washington, USA, March 2, 2008*, pages 16–19, 2008. doi: 10.1145/1353522.1353528. URL https://doi.org/10.1145/1353522.1353528.

[161] Allon Adir, Hagit Attiya, and Gil Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):502–515, 2003. doi: 10.1109/TPDS.2003.1199067. URL https://doi.org/10.1109/TPDS.2003.1199067.

[162] F. Corella, J. M. Stone, and C. M. Barton. Technical report RC18638: A formal specification of the PowerPC shared memory architecture. Technical report, IBM, 1993.

[163] Jade Alglave and Luc Maranget. A diy "seven" tutorial. http://diy.inria.fr/doc/index.html, April 2017.

[164] Jad Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: formal concurrency modelling at arm. Unpublished.

[165] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 190–204, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009838. URL http://doi.acm.org/10.1145/3009837.3009838.

[166] Edward Szpilrajn. Sur l'extension de l'ordre partiel. *Fundamenta Mathematicae*, 16 (1):386–389, 1930. URL http://eudml.org/doc/212499.

[167] Radha Jagadeesan, Alan Jeffrey, and James Riely. Pomsets with preconditions: A simple model of relaxed memory. (OOPSLA), 2020.

[168] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. Modular relaxed dependencies in weak memory concurrency. In Peter Müller, editor, *Programming Languages and Systems*, pages 599–625, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44914-8.

[169] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 378–391, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 158113830X. doi: 10.1145/1040305.1040336. URL https://doi.org/10.1145/1040305.1040336.

[170] Yue Yang, Yue Yang, Ganesh Gopalakrishnan, Ganesh Gopalakrishnan, Gary Lindstrom, and Gary Lindstrom. Formalizing the Java memory model for multithreaded program correctness and optimization. Technical report, University of Utah, 2002.

[171] W. Pugh. The java memory model is fatally flawed. *Concurr. Pract. Exp.*, 12: 445–455, 2000.

[172] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, pages 27–51, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70592-5.

[173] Hans-J. Boehm. How to miscompile programs with "benign" data races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, page 3, USA, 2011. USENIX Association.

[174] T. Riegel, Jeff Preshing, H. Boehm, C. Nelson, and Olivier Giroux. P 0098 r 1 : Towards implementation and use of memory order consume doc. 2015.

[175] Cristian Mattarei, Clark Barrett, Shu-yu Guo, Bradley Nelson, and Ben Smith. EMME: A formal tool for ECMAScript memory model evaluation. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 55–71, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89963-3.

[176] Akshay Gopalakrishnan and Clark Verbrugge. Reordering under the ecmascript memory consistency model. October 2020.

[177] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27:1112–1118, 1978.

[178] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, page 434–442, Washington, DC, USA, 1986. IEEE Computer Society Press. ISBN 081860719X.

[179] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Program. Lang. Syst.*, 8(1):142–153, January 1986. ISSN 0164-0925. doi: 10.1145/5001.5007. URL https://doi.org/10.1145/5001.5007.

[180] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988. ISSN 0164-0925. doi: 10.1145/42190.42277. URL https://doi.org/10.1145/42190.42277.

[181] William W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, Inc., USA, 1992. ISBN 0137671873.

[182] F. Corella, J.M. Stone, and C.M. Barton. *A Formal Specification of the PowerPC Shared Memory Architecture*. IBM Thomas J. Watson Research Division, 1993. URL https://books.google.co.uk/books?id=XYTzrQEACAAJ.

[183] Hagit Attiya and Roy Friedman. Programming dec-alpha based multiprocessors the easy way (extended abstract). In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, page 157–166, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916719. doi: 10.1145/181014.192323. URL https://doi.org/10.1145/181014.192323.

[184] P. Chatterjee and G. Gopalakrishnan. Towards a formal model of shared memory consistency for intel itanium/sup tm/. In *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*, pages 515–518, 2001. doi: 10.1109/ICCD.2001.955081.

[185] A. Adir, H. Attiya, and G. Shurek. Information-flow models for shared memory with an application to the powerpc architecture. *IEEE Transactions on Parallel and Distributed Systems*, 14(5):502–515, 2003. doi: 10.1109/TPDS.2003.1199067.

[186] Nathan Chong and Samin Ishtiaq. Reasoning about the arm weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, MSPC '08, page 16–19, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580494. doi: 10.1145/1353522.1353528. URL https://doi.org/10.1145/1353522.1353528.

[187] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-cc multiprocessor machine code. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 379–391, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583792. doi: 10.1145/1480881.1480929. URL https://doi.org/10.1145/1480881.1480929.

[188] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of Theorem Proving in Higher Order Logics, LNCS 5674*, pages 391–407, 2009. doi: 10.1007/978-3-642-03359-9_27. URL https://doi.org/10.1007/978-3-642-03359-9_27.

[189] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010. ISSN 0001-0782. doi: 10.1145/1785414.1785443. URL https://doi.org/10.1145/1785414.1785443.

[190] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. *SIGPLAN Not.*, 46(6):175–186, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993520. URL https://doi.org/10.1145/1993316.1993520.

[191] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-isa architectural envelope definition, and test oracle, for ibm power multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 635–646, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340342. doi: 10.1145/2830772.2830775. URL https://doi.org/10.1145/2830772.2830775.

[192] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the armv8 architecture, operationally: Concurrency and isa. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 608–621, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837615. URL https://doi.org/10.1145/2837614.2837615.

[193] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. ARMv8-A system semantics: instruction fetch in relaxed architectures (extended version). In *Proceedings of the 29th European Symposium on Programming*, April 2020.

[194] Yatin A. Manerkar, Caroline Trippel, D. Lustig, Michael Pellauer, and M. Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. *ArXiv*, abs/1611.01507, 2016.

[195] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), June 2013. ISSN 0004-5411. doi: 10.1145/2487241.2487248. URL `https://doi.org/10.1145/2487241.2487248`.

[196] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 405–418, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450349116. doi: 10.1145/3173162.3177156. URL `https://doi.org/10.1145/3173162.3177156`.

Appendix A

Armv8-a relaxed memory model

This is the Armv8-a mixed-size axiomatic memory model defined by Christopher Pulte (see §5.4.1).

$$
\begin{array}{rcl}
\mathit{mode} & ::= & \mathsf{ReadAcquire} \\
& | & \mathsf{ReadWeakAcquire} \\
& | & \mathsf{WriteRelease} \\
& | & \mathsf{DMB.SY} \\
& | & \mathsf{DMB.LD} \\
& | & \mathsf{DMB.ST} \\
& | & \mathsf{ISB}
\end{array}
$$

ReadAcquire and WriteRelease correspond roughly to sequentially consistent atomics. DMB and ISB are barriers which are not used by JavaScript's Armv8-a compilation scheme.

$$
\mathit{addr} \quad ::= \quad \alpha \ldots \text{ an infinite set of abstract names}
$$

$$
\begin{array}{rcll}
\mathit{event} & ::= & \{\ \text{ord} & : \mathit{mode} \\
& & \text{index} & : \mathit{nat} \\
& & \text{reads} & : \mathit{list}\ \mathit{byte} \\
& & \text{writes} & : \mathit{list}\ \mathit{byte} \\
\mathit{exec} & ::= & \{\ \text{evs} & : \mathit{set}\ \mathit{event} \\
& & \mathsf{IW} & : \mathit{event} \\
& & \text{program-order } // \ \mathbf{po} & : \mathit{set}\ (\mathit{nat} \times \mathit{event} \times \mathit{event}) \\
& & \text{reads-byte-from } // \ \mathbf{rbf} & : \mathit{set}\ (\mathit{nat} \times \mathit{event} \times \mathit{event}) \\
& & \text{bytewise-coherence } // \ \mathbf{cob} & : \mathit{set}\ (\mathit{nat} \times \mathit{event} \times \mathit{event}) \\
& & \text{address-dependency-pre } // \ \mathbf{addrP} & : \mathit{set}\ (\mathit{event} \times \mathit{event}) \\
& & \text{data-dependency-pre } // \ \mathbf{dataP} & : \mathit{set}\ (\mathit{event} \times \mathit{event}) \\
& & \text{control-dependency-pre } // \ \mathbf{ctrlP} & : \mathit{set}\ (\mathit{event} \times \mathit{event}) \\
& & \text{read-modify-write } // \ \mathbf{rmw} & : \mathit{set}\ (\mathit{event} \times \mathit{event})\ \}
\end{array}
$$

The Armv8-a execution follows the JavaScript mixed-size approach of defining a bytwise reads-byte-from relation between events. The unique components of the Armv8-a execution are the syntactic dependencies address-dependency-pre, data-dependency-pre, control-dependency-pre, and the read-modify-write relation. The address-dependency-pre dependency relates a read event to a subsequent (program-order-later) memory access with an address which is computed using the value of the read event. The data-dependency-pre dependency relates a read event to a subsequent write event which writes a value computed using the value of the read event. The control-dependency-pre dependency relates a read event to a subsequent write event, where there is a conditional control flow instruction program-order-before the write with a conditional which is computed using the value of the read event. The read-modify-write relation relates pairs of successful load/store exclusives. Note that JavaScript memory model does not track address, data, or control dependencies, and therefore the compilation correctness proof does not rely on any contraints in the Armv8-a model which depend on these relations.
Following herd [152], we define relational composition ";" and the identity relation "[A]" on a set A:

$$R; S \triangleq \{\langle A, B\rangle \mid \exists C. \langle A, C\rangle \in R \wedge \langle C, B\rangle \in S\}$$
$$[E] \triangleq \{\langle E, E\rangle \mid E \in A\}$$

Then, $[A]; R; [B]$ is the relation R restricted to elements from A on the left and elements from B from the right:

$$[A]; R; [B] = \{\langle a, b\rangle \mid \langle a, b\rangle \in R \wedge a \in A \wedge b \in B\}$$

Derived auxiliary relations *(with respect to a candidate execution)*

$$\text{same-thread} \mathbin{/\!/} \textbf{sthd} \triangleq \{\langle A, B\rangle \mid \langle A, B\rangle \in \textbf{po} \vee \langle B, A\rangle \in \textbf{po}\}$$
$$\text{reads-from} \mathbin{/\!/} \textbf{rf} \triangleq \{\langle A, B\rangle \mid \exists k. \langle k, A, B\rangle \in \text{reads-byte-from}\}$$
$$\text{coherence} \mathbin{/\!/} \textbf{co} \triangleq \{\langle A, B\rangle \mid \exists k. \langle k, A, B\rangle \in \text{bytewise-coherence}\}^+$$
$$\text{reads-from-internal} \mathbin{/\!/} \textbf{rfi} \triangleq \text{reads-from} \cap \text{same-thread}$$
$$\text{reads-from-external} \mathbin{/\!/} \textbf{rfe} \triangleq \text{reads-from} \backslash \text{reads-from-internal}$$
$$\text{coherence-internal} \mathbin{/\!/} \textbf{coi} \triangleq \text{coherence} \cap \text{same-thread}$$
$$\text{coherence-external} \mathbin{/\!/} \textbf{coe} \triangleq \text{coherence} \backslash \text{coherence-internal}$$
$$\text{bytewise-from-reads} \mathbin{/\!/} \textbf{frb} \triangleq$$
$$\{\langle k, A, B\rangle \mid \exists C. \langle k, C, A\rangle \in \text{reads-byte-from} \wedge \langle k, C, B\rangle \in \text{bytewise-coherence}\}^+$$
$$\text{from-reads} \mathbin{/\!/} \textbf{fr} \triangleq \{\langle A, B\rangle \mid \exists k. \langle k, A, B\rangle \in \text{bytewise-from-reads}\}^+$$
$$\text{from-reads-internal} \mathbin{/\!/} \textbf{fri} \triangleq \text{from-reads} \cap \text{same-thread}$$
$$\text{from-reads-external} \mathbin{/\!/} \textbf{fre} \triangleq \text{from-reads} \backslash \text{from-reads-internal}$$
$$\text{po-same-byte-location} \mathbin{/\!/} \textbf{polocb} \triangleq$$
$$\{\langle k, A, B\rangle \mid \langle A, B\rangle \in \textbf{po} \wedge k \in \text{range}(A) \wedge k \in \text{range}(B)\}$$

$$
\begin{aligned}
\mathsf{R} &\triangleq \{E \mid E.\mathrm{reads} \neq []\} \\
\mathsf{W} &\triangleq \{E \mid E.\mathrm{writes} \neq []\} \\
\mathsf{A} &\triangleq \{E \mid E.\mathrm{ord} = \mathsf{ReadAcquire}\} \\
\mathsf{Q} &\triangleq \{E \mid E.\mathrm{ord} = \mathsf{ReadWeakAcquire}\} \\
&\triangleq \{E \mid E.\mathrm{ord} = \mathsf{WriteRelease}\} \\
\mathsf{DMB.SY} &\triangleq \{E \mid E.\mathrm{ord} = \mathsf{DMB.SY}\} \\
\mathsf{DMB.LD} &\triangleq \{E \mid E.\mathrm{ord} = \mathsf{DMB.LD}\} \\
\mathsf{DMB.ST} &\triangleq \{E \mid E.\mathrm{ord} = \mathsf{DMB.ST}\} \\
\mathsf{ISB} &\triangleq \{E \mid E.\mathrm{ord} = \mathsf{ISB}\} \\
\mathsf{B} &\triangleq \mathsf{DMB.SY} \cup \mathsf{DMB.LD} \cup \mathsf{DMB.ST} \cup \mathsf{ISB}
\end{aligned}
$$

$$
\begin{aligned}
\text{address-dependency} \mathbin{//} \mathbf{addr} &\triangleq [R]; \mathbf{addrP} \\
\text{data-dependency} \mathbin{//} \mathbf{data} &\triangleq [R]; \mathbf{dataP} \\
\text{control-dependency} \mathbin{//} \mathbf{ctrl} &\triangleq [R]; \mathbf{ctrlP}
\end{aligned}
$$

$$
\begin{aligned}
\text{coherence-after} \mathbin{//} \mathbf{ca} &\triangleq \mathbf{co} \cup \mathbf{fr} \\
\text{observed} \mathbin{//} \mathbf{obs} &\triangleq \mathbf{rfe} \cup \mathbf{fre} \cup \mathbf{coe}
\end{aligned}
$$

$$
\begin{aligned}
\text{dependency-ordered-before} \mathbin{//} \mathbf{dob} \triangleq\ & \mathbf{addr} \cup \\
& \mathbf{data} \cup \\
& (\mathbf{ctrl}; [\mathsf{W}]) \cup \\
& ((\mathbf{ctrl} \cup \mathbf{addr}; \mathbf{po}); [\mathsf{ISB}]; \mathbf{po}; [\mathsf{R}]) \cup \\
& (\mathbf{addr}; \mathbf{po}; [\mathsf{W}]) \cup \\
& ((\mathbf{ctrl} \cup \mathbf{data}); \mathbf{coi})) \cup \\
& ((\mathbf{addr} \cup \mathbf{data}); \mathbf{rfi})) \\
\text{atomic-ordered-before} \mathbin{//} \mathbf{aob} \triangleq\ & \mathbf{rmw} \cup (\mathbf{rmw}; [\mathsf{W}]; \mathbf{rfi}; [\mathsf{A} \cup \mathsf{Q}])
\end{aligned}
$$

$$\begin{aligned}\text{barrier-ordered-before} \mathbin{/\!/} \textbf{bob} \triangleq\ & \textbf{po}; [\text{DMB.SY}]; \textbf{po} \cup \\ & ([\text{L}]; \textbf{po}; [\text{A}]) \cup \\ & ([\text{R}]; \textbf{po}; [\text{DMB.LD}]; \textbf{po}) \cup \\ & ([\text{A} \cup \text{Q}]; \textbf{po}) \cup \\ & ([\text{W}]; \textbf{po}; [\text{DMB.ST}]; \textbf{po}; [\text{W}]) \cup \\ & (\textbf{po}; [\text{L}]) \cup \\ & (\textbf{po}; [\text{L}]; \textbf{coi}) \cup\end{aligned}$$

$$\text{initial-ordered-before} \mathbin{/\!/} \textbf{iob} \triangleq \{(\text{IW}, E) \mid E \in \text{EV} \backslash \text{IW}\}$$

$$\text{ordered-before} \mathbin{/\!/} \textbf{ob} \triangleq \textbf{obs} \cup \textbf{dob} \cup \textbf{aob} \cup \textbf{bob}$$

Well-formedness conditions

Let R_k, for some relation $R : \mathit{set}\ (\mathit{nat} \times \mathit{event} \times \mathit{event})$, denote the relation $\{\langle A, B\rangle \mid \langle k, A, B\rangle \in R\} : \mathit{set}\ (\mathit{event} \times \mathit{event})$.

- The initial write is not a release write:

$$\text{IW} \subseteq W \backslash L$$

- In our fragment, reads and writes are disjoint, and reads, writes, and barriers are disjoint:

$$\text{R} \cap \text{W} \cap \text{DMB.SY} \cap \text{DMB.LD} \cap \text{DMB.ST} \cap \text{ISB} = \varnothing$$

- Barriers don't read or write memory:

$$\{\text{range}(E) \mid E \in \text{B}\} = \varnothing$$

- For all locations k, $\textbf{cob}_k$ is a strict total order on $\{W \mid k \in W.\text{writes}\}$ ...
- ... and only includes writes that write to k:

$$\forall k, W, W'.\, \langle W, W'\rangle \in \textbf{cob}_k \implies k \in W.\text{writes} \cap W'.\text{writes}$$

- coherence is a strict partial order.

- read-modify-write, address-dependency, data-dependency, and control-dependency are subsets of program-order:

$$\mathsf{rmw} \cup \mathsf{addr} \cup \mathsf{data} \cup \mathsf{ctrl} \subseteq \mathsf{po}$$

Candidate execution validity

Single-copy atomicity
$(\mathsf{rf}; \mathsf{fr})$ irreflexive

Coherence/internal
$\forall k.(\mathsf{polocb}_k \cup \mathsf{frb}_k \cup \mathsf{cob}_k \cup \mathsf{rbf}_k)$ acyclic

External
ob acyclic

Exclusives/Atomic
$\mathsf{rmw} \cap (\mathsf{fre}; \mathsf{coe}) = \varnothing$